Towards Open Services on the Web - A Semantic Approach

Thesis

THE OPEN UNIVERSITY, UK

Towards Open Services on the Web - A Semantic Approach

by

Dipl.-Inform. Maria Maleshkova

Thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy in Computer Science

Knowledge Media Institute (KMi)

March 27, 2014

Preface

"What doesn't kill you makes you stronger."

- Friedrich Nietzsche

I am not sure if Nietsche was working on his thesis when he came up with this statement but he might just as well have been. We had a PhD adaptation of this quote to "What doesn't kill you makes you fatter", which reflects quite adequately the correlation between paper deadlines and consumption of custard donuts or chocolate.

The three years that I spent at KMI have been the most dramatic, fulfilling, exhausting, rewarding and intense in my life so far. For better or for worse, the PhD experience has been a life-changing one. Unfortunately no one can be told what being a PhD student is, you have to see it for yourself. In the mean time I strongly believe that one is awarded the title not only for the research contributions but also for actually going through the process and gaining the experience and knowledge that are required to reach the point of submitting the written thesis.

As a computer scientist, I have always been very fascinated by the idea that software components that implement a certain algorithm, or provide access to a sensor, or simply expose data, can be accessed remotely by anyone in the world and be integrated in new applications and solutions. Therefore, conducting research in the field of Web services and Web APIs has not only presented a new and engaging challenge but also connected with my personal interests. I am very thankful to all of my colleagues from the Semantic Web Services Group at KMI but especially to my supervisors – Carlos Pedrinaci and John Domingue. John saw the researcher in me before anyone else did and Carlos helped me become the researcher that I am today. They have undoubtedly played a central role in this journey and have been very supportive, giving me a lot of opportunities to develop and pursue my research interests. I have always been able to rely on their guidance and have learned a lot, starting with basic technical skills, building up to reviewing the work of fellow researchers and writing high-quality papers. However, most importantly, through them I learned what passion for science and dedication to work are. They taught me the right attitude towards research and have strongly influenced my professional development.

I was very lucky to have great colleagues at KMI, who were a very important part of my life both on professional and personal level. I really enjoyed working with Laurian, Dave, Dong, Ning and Jacek. Laurian taught me how to detangle impossible to understand code and how to hack applications so that they work (ignoring most of the things that I was taught about software development at the university). I still miss our morning coffee sessions with Dave, which were mainly focused on complaining about the general difficulties of life. This moaning sessions,

contrary to expectations, always encouraged me to be productive and optimistic. The collaboration with Dong and Ning has taught me a lot as well and, even though, we were not close fiends I was very fortunate to have them in our research group. I will not miss the passionate discussions with Jacek and his “Give me an example” statements (well maybe I will miss them a little) but working with him has been great and gave me a new perspective on caring about detail, line of argumentation, and structuring research approaches in general. I would also like to thank Simone and Bruno for their collaboration on some of the more practically-oriented tasks. I also need to mention my dear Miri and Hassan, who were always up for a friendly coffee and a chat. KMI has been my home during my PhD years and I feel very privileged to have been a part of a team of so talented and hard-working researchers.

I am also very thankful to Rudi Studer, who gave me the opportunity to start doing research in the first place. After completing my diploma thesis at AIFB, he considered that I had the potential to become a researcher and offered me a PhD position in his group. I have very much enjoyed working at AIFB during the past one and a half years.

Finally, this would never have been possible without the support of my amazing family. They have encouraged me and comforted me every step of the way. Some things in life are much easier when you know that no matter what happens someone is there for you, ready to be on your side no matter what. For me, this has always been my family and I think that this security has given me the mindset to strive for the top and to live a life where no limits can be set. There are no words to express how grateful I am for that.

THE OPEN UNIVERSITY, UK

Abstract

Knowledge Media Institute (KMi)

Doctor of Philosophy in Computer Science

by Dipl.-Inform. Maria Maleshkova

The World Wide Web (WWW) has significantly evolved since it was first released as a publicly available service on the Internet, developing from a collection of a few interlinked static pages to a global ubiquitous platform for sharing, searching and browsing dynamic and customisable content, in a variety of different media formats. It plays a major role in the lives of individuals, as a source of information, knowledge and entertainment, as well as in the way business and communication are done within and between companies. This transformation was triggered by the ever-growing number of users and websites, and continues to be supported by current developments such as the Social Web, Linked Data, and Web APIs and Services, which together pave the way for the Web as a dynamic data environment.

The work presented in this thesis aims to contribute to a more integrated Web, where services, data and Web content can be seamlessly combined and interlinked, without having to deal with the intricacies of the separate data sources or the specific technology implementations. The vision of *Open Services on the Web* aims to facilitate the unified use of Web APIs, Web Services and Linked Data sources, so that users can retrieve data without differentiating whether its source is a website, a Web API or even a mashup. However, before this can be achieved, there are a number of problems that need to be addressed. In particular, the integrated and unified handling of services, and especially Web APIs, is very challenging because of the heterogeneous landscape of implementation approaches, underlying technologies and forms of documentation. In particular, in the context of Web APIs, the main limitations are caused by the fact that currently documentation is commonly provided directly in HTML, as part of a webpage, which is not meant for automated machine processing of the service properties, in contrast to XML, for example. This situation is aggravated by the fact that Web APIs are proliferating quite autonomously, without adhering to particular guidelines and specifications. This results in a wide variety of description forms and structures, accompanied by a range of diverse underlying technologies, forcing developers to individually interpret the documentation, and carry out complicated and tedious development work. The result is the implementation of individual custom solutions that are rarely reusable and have very low support for interoperation.

We contribute towards achieving the vision Open Services on the Web by tackling some of these challenges and supporting the wider, integrated and more automated use of Web APIs. In particular, we present a thorough analysis of the current state of Web APIs, giving the results of two Web API surveys. We use the collected data in order to draw conclusions about occurring practices and trends, and common API characteristics. The results provide essential input for acquiring a real-world view on Web APIs, for identifying key service properties, for determining best practices, for pointing out difficulties and implementation challenges, and for deducing a baseline for the support that any solution approach needs to provide. The so gathered details are used for developing a shared formal model for describing, modelling and annotating Web APIs, which serves as the basis for decreasing the level of manual effort, involved in completing common service tasks, and provides a unifying overlay on top of the heterogeneous API landscape. This shared model – the *Core Service Model* captures all essential API characteristics, thus providing common grounds for developing support solutions in the context of using Web APIs, but also enables a unified view over traditional Web services and APIs, facilitating their interoperable handing and enabling the reuse of existing Web service approaches and solutions.

The work presented here aims to contribute to a more interoperable and automated use of Web APIs, however, our focus is mainly on providing support for the completion of the invocation and authentication tasks. These tasks are essential, since the exploitation of an API is dependent on how well and how easy the process of actually sending an input, making a call, and receiving a corresponding output, can be completed. In particular, we address this by defining extensions to the Core Service Model, which are targeted specifically at supporting invocation and authentication. These extensions are captured in the form of two formal models - the *Web API Grounding Model* and the *Web API Authentication Model*. We describe the support provided by the individual model elements, we demonstrate how they can be used to make annotations, and we back up the presented approaches by actual implementations in the form of invocation and authentication engines, which take as input annotated APIs and use these to automatically complete the invocation and authentication processes.

Finally, we recognise the need for encouraging the adoption of the newly developed models and we address this with the help of tools and annotation approaches. In particular, we introduce *SWEET*, which is a web application that enables users to annotate HTML Web API documentation. It hides formalism complexities behind a graphical interface and reduces the annotation effort to selecting a particular property and linking it to the corresponding model element. Furthermore, we support the semi-automated annotation process, by providing integrated ontology and semantic entity search, and directly classifying the functionality of an API. The result is an annotated HTML documentation, which can be republished on the Web, stored in a service repository or be converted to a semantic RDF description that can be handled and processed alongside existing semantic Web service descriptions.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

The World Wide Web (WWW) has undergone significant changes since it was first launched in 1991. It has evolved from an infrastructure for static content of pages consumed by individual users to a communication platform where people, organisations, companies, and devices alike offer, consume and synthesise content and services on a massive scale. The initial founding concept of providing individual pages, containing formatted text and simple media, that are interlinked with further pages, has been adapted and further developed in order to provide the basis for the building of communities and social networks, exposing businesses and consumers to new markets, and enabling the creation of applications and services, which offer much more complex functionalities than the initially envisaged static HTML content. The rise of Web 2.0 [O'R09], the Social Web [Gru07] and the Semantic Web [BLHL01, SHBL06] have paved the way for more recent developments such as the growing popularity of Web applications and APIs [MPD10a] and the wider adoption of the Linked Data principles [BHBL09].

In particular, the value of Web platforms and applications is no longer restricted to only directly making content available to users but is also in providing access to resources and functionality through publicly available APIs that do not have a graphical user interface but are rather designed for direct machine consumption[1]. The current trend of providing access to data in a programmable way and exposing resources, in a format that is not meant for human users but is rather targeted towards computer interpretation and processing, is supported by popular social platforms, such as Facebook, Google, Flickr, YouTube and Twitter, and online service providers, such as Google Maps, Google Search and eBay. These online platforms offer public APIs enabling third parties to combine and reuse heterogeneous data coming from diverse software component interfaces, i.e. services, in data-oriented service compositions called mashups [SET09]. Similarly, more and more data holders such as government and public organisations, as well as private institutions, expose their data on the Web as Linked Data [BHBL09], thus contributing to

[1] 75% of Twitter traffic goes directly through its Web API, source Programmable Web – Twitter Reveals: 75% of Our Traffic is via API (3 billion calls per day), `goo.gl/BtkM6U`, visited April 2012.

a Web of Data [BHBL09] that lays the foundation for machine-oriented retrieval and processing of different types of data.

As a result, the current popularity of Web APIs and the increasing importance of Linked Data provide an opportunity not only for the machine-based retrieval and manipulation of diverse resources but also for the building of versatile applications based on combining heterogeneous data and manipulated by the means of openly exposed processing functionalities. The vision of *Open Services on the Web* aims to facilitate the unified and integrated use of Web APIs, Web Services and Linked Data sources. However, before this can be achieved, there are a number of problems that need to be addressed first. The work presented in this thesis aims to contribute precisely towards a Web, which is based on more dynamic content and seamlessly integrates data and services, which can be transparently discovered, composed and executed by the computer on behalf of its user.

1.1 Thesis Motivation

The world of services on the Web is increasingly dominated by Web applications and APIs, which seem to be preferred over "traditional" Web services [ACKM04] in the context of developing Web applications and mashups, by accessing resources available over the Web. Web services, based on WSDL [W3C07a] and SOAP [W3C07b], have played and, without a doubt, will continue to play a major role in the development of loosely coupled component-based systems within and between enterprises [Bel08]. However, the past few years have been marked by a trend towards a simpler approach for developing and exposing programmable interfaces, moving away from the rather complex WS-*specification stack [CFNO04]. Instead, current Web service providers are inspired by a technology that is based on adopting the original design principles of the World Wide Web [BL99] to the world of services on the Web. The result is the current proliferation of Web APIs[2] that rely directly on the interaction primitives provided by the HTTP protocol, with data payloads transmitted directly as part of the HTTP requests and responses. Therefore, Web APIs, also referred to as RESTful services [RR07], when conforming to the REST architectural principles [Fie00], are characterised by their relative simplicity and their natural suitability for the Web. On the basis of this simple technology stack, providers offer public APIs, which enable access by 3rd parties to some of the resources they hold, thus enabling the direct retrieval and processing of data but also the building of applications, which might use single or aggregated data sources.

Despite their growing importance, Web APIs are still facing a number of limitations. In contrast to traditional Web services, whose processing relies on the information provided in the WSDL

[2]The number of Web APIs and existing mashups at `http://www.programmableweb.com` has progressively increased during the past seven years.

files [W3C07a] and is supported by a stack of specifications, the development of Web APIs has evolved in a rather autonomous way. In fact, while the term "Web Service"[3] is quite clearly defined [Dai12], Web APIs still lack a broadly accepted definition. Currently the term "Web API" has a general, sometimes even controversial, meaning and is used for depicting HTTP-based interfaces, frequently being inconsistent about the specific technical and design underpinnings (see Chapter 7).

Furthermore, the majority of the APIs[4] are exposed and described only through human-oriented documentation, which is useful only to a human developer and conforms to no established guidelines or specifications. This results in a wide variety of documentation forms and structures, which have to be individually read and interpreted. Therefore, currently all related tasks, such as finding suitable services, invoking them or composing them into mashups, rely solely on the information provided in the HTML documentation, which is not meant for automated machine interpretation and has to be processed manually.

In addition to the lack of descriptions that enable a certain degree of automation, only about a third of the APIs are currently conforming to the REST principles, while the majority ignore these best-practices and define interfaces in terms of operations instead of resources (see Chapter 6). This results in a variety of underlying technologies, used to implement APIs, forcing developers to build individual and custom software implementations. The result is a set of solutions that have low interoperability and low levels of reuse. Therefore, current common practices in using Web APIs are time and effort-consuming and will not scale in the context of the growing number of available Web APIs [MPD10a]. Finally, there is still lack of support for the unified handling of both "traditional" Web services and Web APIs – the two types of services are stored in separate directories, commonly use different task automation approaches and are rarely deployed in integrated solutions.

In summary, the main issues related to using Web APIs are the following:
1. There is **no shared, widely accepted definition of what a Web API is**. This results in a lack of clarity and misunderstandings on the conceptual as well as the specific technology implementation level. Furthermore, the gained experience is restricted to individual APIs and the produced solutions have no interoperability.
2. The majority of the Web API **descriptions are human-oriented**, given directly in HTML as part of webpages. This requires manual search, processing and interpretation.
3. The completion of common service tasks such as discovery, composition and invocation, as well as the maintenance and reuse of developed solutions, requires **extensive manual effort and processing**.

[3] `http://www.w3.org/2002/ws/`, last retrieved April 2012.
[4] Throughout this thesis we use the terms "APIs", "services" and "Web APIs" interchangeably.

4. The lack of common guidelines and specifications result in **heterogeneity** of the documentation. This in turn results in the need to individually handle each API and produce custom client solutions with low potential for reuse, thus hampering the scalability and cost-efficiency of the API use.
5. There is very limited support for **unified handling of "traditional" Web services and Web APIs**.

The work presented here addresses all of the identified problems, focusing especially on enabling more integrated and automated Web API use through supporting the automation of the invocation and authentication tasks. In particular, we distinguish between issues related to the syntactical structuring of the documentation, such as the particular format, contained elements, hierarchical structure of the elements, element types, etc., and the semantics of the API, such as the type of service provided, type of input and output, functionality of the operations. This distinction is important since the different challenges can be addressed with different solutions. For example, the second point from the list above relates to the lack of a common syntactical structure for describing Web APIs, while the third one is closely related to the second but also addresses the lack of semantic information – for instance, without having an annotation about the functionality of the API, the developer still has to select a suitable API manually. To this end, our work focuses on the provisioning of a description model that enables the enhancement of existing documentation with semantic annotations. This requires the addressing of existing challenges on both the syntactic and semantic level. The following section lists the research questions that are formed in order to address the identified key problems.

1.2 Research Questions

This thesis aims to contribute towards supporting the wider and more integrated use of Web APIs. Therefore, it focuses on decreasing the level of manual effort involved in completing common service tasks, which is achieved on the basis of addressing the need for a shared model for describing Web APIs. Given the breadth and depth of all the problems raised above, we emphasise mainly on aiding invocation and authentication, and thus enabling more automated Web API use. The need for encouraging the adoption of the newly developed model with the help of tools and annotation approaches is also addressed by the conducted work.

Overall, the presented research contributes towards answering the following main research question:

How can we enable Open Services on the Web?

In particular, we want to explore how the unified and integrated use of Web APIs, Web Services and Linked Data sources can be enabled. Furthermore, we want to identify solutions and approaches that contribute towards enabling a Web, which is based on more dynamic content and seamlessly integrates data and services, which can be transparently discovered, composed and executed by the computer on behalf of its user.

The main research question covers a broad area of possible directions of work. The work presented here is focused on addressing issues related to Web APIs in the context of Open Services on the Web, for it is the area which has least been studied in the literature, is least supported through tools and approaches, and yet is also the most popular nowadays. Therefore, in order to specify the particular scope of the conducted research, the above question is divided into several sub-questions. These are as follows:

RQ1: ***What are the common Web API characteristics?***
Given the current heterogeneity of the Web API landscape, it is not easy to directly grasp what marks out an API and what are all the common features that characterise an API. Currently, the API elements described in the documentation differ and they are captured in varying levels of detail. Furthermore, we take a pragmatic approach, aiming to gather a clear picture of the real-world state of Web APIs and provide incremental support and solutions. To this purpose we want to explore what API descriptions look like, what information they provide, what API elements are given, how they are structured, etc. Before any progress can be made towards supporting the use of Web APIs, first we need to become aware of the current state of Web APIs and gain a deeper understanding of how they are exposed on the Web and what the main problems surrounding their development and use are.

RQ2: ***How to describe Web APIs?***
In particular, we want to identify possible solutions for describing Web APIs in a way such that we cover the majority of the currently existing APIs, support the unified handling of APIs and "traditional" services, and enable the adoption of approaches and solutions devised as part of research on Web services. A more detailed list of needs and requirements is indeed presented in Chapter 7, where we cover the work we have carried out in this regard. However, the main objective here is to explore how Web APIs can be described in such a way as to contribute towards achieving the vision of Open Services on the Web.

RQ3: ***How to enable a more automated Web API use?***
Since currently most of the work related to using Web APIs has to be completed manually, it is important to determine how different tasks, such as discovery, composition and invocation, can be performed with a higher level of automation. In particular, this requires the analysis of

description details, which need to be part of the Web API description, in order to support more automated API use.

RQ4: ***How to support the adoption of the new service model?***
Given a formalism for the description of Web APIs, developers need to be supported by tools that make the creation of Web API descriptions easier. This includes the provisioning of extensive tool support but also automating as much as possible the process of creating Web API descriptions.

1.3 Contributions

This thesis encompasses research on enabling Open Services on the Web through supporting the use of Web APIs. Overall, the work presented here aims to contribute towards a more integrated Web, where Web services, data and Web content can be seamlessly combined and interlinked, without having to differentiate between separate data sources or specific technology implementations. This is achieved by combing semantic approaches, Linked Data principles and fundamental Web technologies, including URIs and HTTP, and applying these on services on the Web, in order to enable the creation of Web API descriptions that can serve as a basis for automating API-based interactions and integration of resources. Bearing this context in mind, this thesis makes the following contributions:

Contribution 1: *A thorough analysis of the current state of Web APIs.*

Before any significant impact and improvement can be made to current Web API practices and technologies, we need to reach a deeper understanding of these. This involves, for instance, figuring out how current APIs are developed and exposed, what kind of descriptions are available, how they are represented, how rich these descriptions are, what is the existing tooling and support, etc. It is only then that we shall be able to clearly identify deficiencies and realise how we can overcome existing limitations, and how much of the available know-how on Web services can be applied.

To date, there is no clear information on the current state of Web APIs and there are no available surveys that reflect on different API features. Therefore, this thesis includes two Web API studies that capture common characteristics and provide a basis for reaching conclusions on the common practices and technologies used when publishing APIs. The results of these surveys directly contribute to understanding existing challenges and are a basis for devising solutions and supporting mechanisms. We take a very pragmatic approach and base the here developed solutions on the current real-world state of APIs, instead of relying on assumptions or theoretical guidelines. In particular, the analysis of the current state of Web APIs lays the foundation for the definition of a common description model. Furthermore, it provides insights about details

that are required as part of the API description, in order to be able to complete common service tasks without the need of a human developer to be engaged in the process.

Contribution 2: *Definition of a Web API* – a term clearly describing what should be taken as a shared understanding when referring to Web APIs.

Despite the current proliferation of Web APIs and the frequent use of the phrase, it is often not clear what the implied underlying principles or technologies are. This is due to the fact that currently there is no specification or shared understanding about the characteristics that mark out a Web API. Therefore, given the current heterogeneity of the world of Web APIs, it is necessary to introduce a unifying definition that can contribute towards a common understanding about APIs but also enforce some shared underlying concepts that can serve as a foundation for developing an API description model and approaches for supporting API use. We develop this definition of a Web API by relying on the survey results from Contribution 1.

Contribution 3: *A core service model*, capturing common API characteristics and providing a basis for supporting the automation of common service tasks.

Based on the results of the Web API analysis we are able to deduce a description model that captures common service characteristics, but also provides high coverage in the context of the diversity of the forms and structure of the available documentation. Given the current autonomous proliferation of Web APIs, the description model is applied on top of existing HTML documentation by enhancing it with specific annotations and does not require the creation of new descriptions from scratch. Furthermore, the model serves as a basis for providing a higher level of automation to common service tasks, by including core service elements that are required for completing these tasks. Finally, the Web API description model also enables the reuse and adaptation of the wealth of research done in the context of Web services and Semantic Web Services. Therefore, it provides common grounds for the unified handling of "traditional" Web services and Web APIs.

Contribution 4: *Web service model extensions for supporting automated invocation and authentication*.

As already mentioned, this thesis focuses on supporting Web API use through enabling a higher level of automation of certain service tasks. Therefore, in addition to providing a shared service model, the conducted work also delivers extensions to the description model that are especially targeted at supporting tasks that are key for facilitating API use – invocation and authentication. The presented model extensions are based on the analysis of the current state of Web APIs but also take into consideration existing description frameworks and approaches in the context of Web services. They can be used in combination with the introduced Web API description model or independently of it and can be extended to accommodate further properties that might be necessary for certain use cases.

Contribution 5: *Support for creating Web API descriptions*, in the form of an annotation tool and task-assisting solutions such as annotation recommendation mechanisms.

In order to be able to practically apply the developed Web API description model, users are provided with annotation support in the form of a Web application tool and mechanisms for assisting the completion of individual tasks along the process of creating API descriptions. The tool has all functionalities necessary for manually annotating Web APIs, while further user assistance is given by automatically determining the type of functionality that the API provides or enabling ontology search for suitable annotations.

Each of the contributions, including the description model but also the supporting tools and annotation solutions, have been thoroughly evaluated.

The contributions listed here address directly problems of the current status-quo in searching for suitable Web APIs and developing client applications that facilitate their use. They pave the way for the development of Open Services on the Web, which are a part of a more dynamic and integrated Web, where Web services, data and Web content can be seamlessly combined and interlinked, without having to differentiate between the separate data sources or the specific technology implementations.

1.4 Thesis Overview

This thesis consist of twelve chapters, which are grouped into four main parts – Introduction, Context and Related Work, Supporting Open Services on the Web, and Evaluation and Conclusions. We introduce the contributions listed in the previous section in Part III: Supporting Open Services on the Web. The thesis is structured as follows:

Part I: Introduction
This part introduces the context of our work and describes the current challenges faced by Web APIs. In addition, the first chapter includes a list of the research questions as well as an overview of the contributions made, while working towards providing solutions to the posed questions. Chapter 2 describes the research approach that we followed in enabling better Web API use, in particular, through supporting the automation of the invocation and authentication tasks.

Part II: Context and Related Work
The second part sets the context of our work and focuses on describing similar and related research in the filed. In particular, this part is divided into three chapters. Chapter 3 gives details on existing Semantic Web Service and Web API description approaches and points out some of the limitations and drawbacks that they have. Chapter 4 provides an overview over current Web

API invocation and authentication methods and solutions, while Chapter 5 focuses on tools and implementations that provide tagging and annotation support.

Part III: Supporting Open Services on the Web

This part describes our main contributions, starting with Chapter 6, which focuses on providing details on the results of the two Web API surveys and the conclusions that we can draw regarding current practices and technologies, and common properties. This chapter also describes the survey application that we implemented in order to be able to conduct further similar studies.

Chapter 7, 8 and 9 are structured very similarly. Each chapter starts with a set of requirements that need to be met by the developed model, followed by a list of design principles and decisions. The core part of each of the chapters describes the model, its individual properties and gives an example of how it can be used to make annotations. In particular, Chapter 7 gives details on the core service model, its main classes and how it can support the annotation of the majority of the APIs. Chapter 8 presents extensions to the core model, which are specifically targeted at supporting the automation of invocation, while Chapter 9 includes details on the design and content of the Web API Authentication Model, which, as the name suggests, enables the capturing of authentication-relevant characteristics.

This part is concluded by Chapter 10, which gives information about our tool and the developed approaches for supporting the creation of semantic Web API descriptions, based on the previously introduced core service model. This chapter includes details on the different versions and functionalities of SWEET – a web application that supports the creation of semantic Web API descriptions. We also describe our approaches for classifying the type of API and the integrated ontology search, which assist the user in making API annotations.

Part IV: Evaluation and Conclusions

The fourth part of the thesis concludes our work. Chapter 11 includes the evaluation of the introduced models. In particular, we determine how well they conform with the defined design requirements, what coverage they provide, and what is the level of support in terms of enabling the automation of the invocation and authentication tasks. This chapter also provides the results of the evaluation of SWEET and the two developed Web API classification approaches. In Chapter 12 we discuss our conclusions and contributions, and point out future work.

1.5 List of Publications

Most of the chapters in this thesis are based on work that has been presented as part of conference contributions[5] or published in the form of book chapters:

- Chapter 6 on exploring the current state of Web APIs in partially based on:
  - Maleshkova, M., Pedrinaci, C. and Domingue, J. (2010) Investigating Web APIs on the World Wide Web, European Conference on Web Services (ECOWS), Ayia Napa, Cyprus, which describes the results of the first Web API survey.

- Chapter 7 is based on a number of publications that describe the Minimal Service Model and its relationship to the iServe service repository:
  - Pedrinaci, C., Maleshkova, M., Zaremba, M. and Panahiazar, M. (2012) Semantic Web Services Approaches, in eds. Alistair Barros, Daniel Oberle, Handbook of Service Description: USDL and its Methods, Springer
  - Pedrinaci, C., Kopecky, J., Maleshkova, M., Liu, D., Li, N. and Domingue, J. (2011) Unified Lightweight Semantic Descriptions of Web APIs and Web Services, Workshop: W3C Workshop on Data and Services Integration, Bedford, MA, USA
  - Kopecky, J., Vitvar, T., Pedrinaci, C. and Maleshkova, M. (2011) RESTful Services with Lightweight Machine-readable Descriptions and Semantic Annotations, in eds. Erik Wilde, Cesare Pautasso, REST: From Research to Practice, Springer
  - Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J. and Domingue, J. (2010) iServe: a Linked Services Publishing Platform, Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference
  - Pedrinaci, C., Lambert, D., Maleshkova, M., Liu, D., Domingue, J. and Krummenacher, R. (2010) Adaptive Service Binding with Lightweight Semantic Web Services, in eds. Schahram Dustdar and Fei Li, Service Engineering: European Research Results, Springer
  - Domingue, J., Pedrinaci, C., Maleshkova, M. and Krummenacher, B. (2011) Fostering a Relationship Between Linked Data and the Internet of Services, in eds. John Domingue, Alex Galis, Anastasius Gavras, Theodore Zahariadis, Dave Lambert, Frances Cleary, Petros Daras, Srdjan Krco, Henning Müller, Man-Sze Li, Hans Schaffers, Volkmar Lotz, Federico Alvarez, Burkhard Stiller, Stamatis Karnouskos, Susana Avesta, Michael Nilsson, Future Internet: Achievements, Directions and Promises, Springer

- Chapter 8 includes publications on the Web API Grounding model, as well as published work on OmniVoke:
  - Maleshkova, M., Pedrinaci, C., Li, N., Kopecky, J. and Domingue, J. (2011) Lightweight Semantics for Automating the Invocation of Web APIs, IEEE International Conference on Service Oriented Computing & Applications (SOCA 2011), Irvine, California, USA

[5] In detail, the publications include 4 book chapters, 5 conferences, 7 workshops, and 4 demos and posters.

- Li, N., Pedrinaci, C., Maleshkova, M., Kopecky, J. and Domingue, J. (2011) OmniVoke:A Framework for Automating the Invocation of Web APIs, Fifth IEEE International Conference on Semantic Computing, Stanford University, Palo Alto, CA, USA
- Li, N., Pedrinaci, C., Kopecky, J., Maleshkova, M., Liu, D. and Domingue, J. (2011) Towards Automated Invocation of Web APIs, Poster at 8th Extended Semantic Web Conference
- Liu, D., Li, N., Pedrinaci, C., Kopecky, J., Maleshkova, M. and Domingue, J. (2011) An Approach to Construct Dynamic Service Mashups using Lightweight Semantics, Workshop: The 3rd International Workshop on Lightweight Integration on the Web (ComposableWeb 2011) at The 11th International Conference on Web Engineering (ICWE 2011)

- Chapter 9 is based on one main publication on the Web API Authentication Model
  - Maleshkova, M., Pedrinaci, C., Domingue, J., Alvaro, G. and Martinez, I. (2010) Using Semantics for Automating the Authentication of Web APIs, International Semantic Web Conference (ISWC), Shanghai, China

- Chapter 10 encompasses a series of publications that describe SWEET, its functionalities and application scenarios:
  - Maleshkova, M., Pedrinaci, C. and Domingue, J. (2010) Semantic Annotation of Web APIs with SWEET, Workshop: 6th Workshop on Scripting and Development for the Semantic Web at Extended Semantic Web Conference, Heraklion, Greece
  - Maleshkova, M., Kopecky, J. and Pedrinaci, C. (2009) Adapting SAWSDL for Semantic Annotations of RESTful Services, Workshop: Beyond SAWSDL at OnTheMove Federated Conferences & Workshops, Vilamoura, Portugal
  - Maleshkova, M., Pedrinaci, C. and Domingue, J. (2009) Supporting the Creation of Semantic RESTful Service Descriptions, Workshop: Service Matchmaking and Resource Retrieval in the Semantic Web (SMR2) at 8th International Semantic Web Conference, Proceedings of ISWC '09, Washington D.C., USA
  - Maleshkova, M., Zilka, L. and Pedrinaci, P. (2011) Cross-Lingual Web API Classification and Annotation, Workshop: The Multilingual Semantic Web at 10th International Semantic Web Conference, Proceedings of ISWC 2011, Bonn, Germany
  - Duke, A., Stincic, S., Davies, J., Lecue, F., Mehandjiev, N., Pedrinaci, C., Maleshkova, M., Domingue, J., Liu, D. and Alvaro, G. (2010) Telecommunication mashups using RESTful services, ServiceWave 2010
  - Maleshkova, M., Pedrinaci, C. and Domingue, J. (2009) Semantically Annotating RESTful Services with SWEET, Demo at 8th International Semantic Web Conference, Washington D.C., USA
  - Maleshkova, M., Gridinoc, L., Pedrinaci, C. and Domingue, J. (2009) Supporting the Semi-Automatic Acquisition of Semantic RESTful Service Descriptions, Poster at ESWC 2009

- Maleshkova, M., Gridinoc, L., Pedrinaci, C. and Domingue, J. (2009) Semi-Automatic Acquisition of Semantic RESTful Service Descriptions, Poster at 2nd STI International Offsite, Crete, Greece

Chapter 2

Approach

This chapter introduces the approach followed in this thesis. In particular, we aim to pave the way for a Web, which is based on a dynamic and integrated use of content, which can come from both services on the Web and the Web of data alike, so that users can retrieve data without differentiating whether its source is a website, a Web API or even a mashup. The approach that we follow towards achieving this goal is by enabling a more automated and interoperable Web API use. To this purpose, we employ declarative descriptions of software component interfaces, which are accessible over standard Web technologies (e.g. URIs and HTTP). Furthermore, we use semantics as the basis for building an abstraction level over the existing heterogeneity in terms of Web API documentation forms and structures, and the foundation for a higher level of automation through supporting the development of reasoning algorithms based on semantic entities. Finally, we also provide tool support for creating Web API descriptions enriched with semantic metadata.

This chapter consists of two sections: We start with Section 2.1, which introduces the three main postulates, which we follow. Section 2.2 describes in detail each of the parts of the introduced approach.

2.1 Introduction

Our approach is based on taking into consideration three main postulates. First, given the current autonomous proliferation of Web APIs, we advocate a non-invasive solution towards describing them – i.e. we do not try to enforce a new description format and language, which would require providers to rewrite all their documentation and partially adjust the implementations, which would undoubtedly also affect all client implementation. A large number of the existing Web API description formalisms and implementation infrastructures take an idealistic view on

the current state of Web APIs and assume conformity to a certain set of principles, such as REST (see section 3.4). Instead we consider the current state of Web APIs as they are, with their heterogeneity and lack of uniform solutions. Therefore, we explore ways of enhancing existing HTML-based documentation with metadata, including the incremental structuring of the documentation by identifying service properties such as operations, inputs and outputs, so that they can be automatically processed, and the subsequent enrichment of these properties with semantic information, which serves as the basis for a more automated completion of common Web API tasks.

Second, we take a lightweight semantic approach. Research on Semantic Web Services (SWS) has been devoted to reduce the amount of manual effort required for manipulating Web services. The main idea behind this research is that tasks such as discovery, negotiation, composition and invocation of Web services can have a higher level of automation and achieve better results, when services are enhanced with semantic descriptions of their properties. Even though Web APIs are widely used, they are currently facing the same limitations as "traditional" Web services, in addition to further problems arising from the lack of standardisation and the availability of only textual documentation. Therefore, we use lightweight semantics to contribute towards reducing the amount of manual effort required for using Web APIs. We say that the semantics are lightweight because instead of relying on extensive description frameworks that employ complex ontologies and reasoning techniques, we follow the principle of "A Little Semantics Goes a Long Way" [Hen97] and stick to a simple description model that is easier to use for making annotations but still provides basic support for performing common service tasks.

Third, we aim towards providing a unified view over both "traditional" Web services and APIs, not only to enable their integrated use in common solutions but also to benefit from the richness of research in the SWS area. WSDL-based services and APIs are treated in a completely disjoint way, however, in certain use cases, especially in the context of the development of end-user applications and service compositions, support for shared handling would be extremely useful. Furthermore, research in the area of SWS has been quite prolific and it would be very useful to directly adopt some of the existing solutions, instead of starting from scratch.

2.2 Methodology

Our methodology includes three main parts (visualised in Figure 2.1). First, **Field Analysis** is conducted in the form of a thorough study of the current state of Web APIs, focusing on analysing common characteristics and deducing trends and followed practices. This step is necessary in order to gain a realistic overview of the current state of Web APIs.

Second, the gathered results are used as input to the **Solution Development**. The main objectives of this part are twofold. First, it includes the development of a conceptual solution in terms of a model for the description of Web APIs. The model captures both the individual API parts such as operations, inputs and outputs, and provides support for adding metadata in the form of semantic annotations, which, for example, describe the particular functionality of the API or the types of data that are consumed and produced. Second, the solution development also includes the design and implementation of tooling for creating semantic Web API descriptions. This includes tools, which hide formalism complexities behind a user interface, and automated techniques, which reduce the manual effort required for service annotation, for example, by suggesting suitable service metadata. In this way, the conceptual solution for describing Web APIs can be directly put into practice by using the implemented tools.

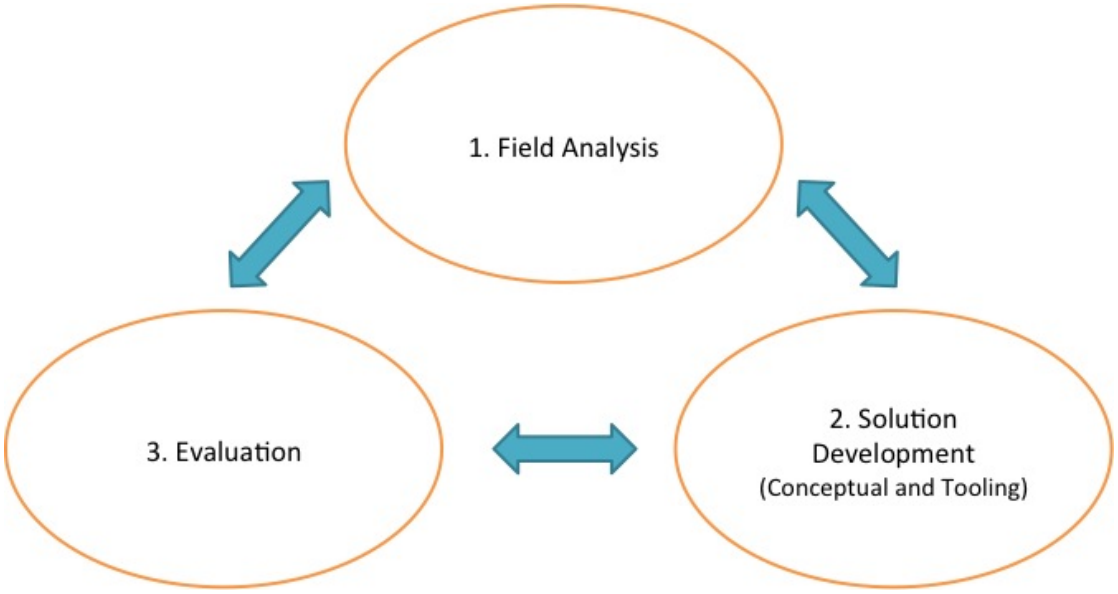


FIGURE 2.1: Methodology Overview

The third and final part of the approach is the **Evaluation**, which is divided into two steps, corresponding to the two objectives of the solution development. First, we evaluate the conceptual solution. Relevant metrics that we consider are conformity of the model with the defined design requirements, the coverage that it provides, and the level of support in terms of enabling the automation of the invocation and authentication tasks. Second we evaluate the developed tooling – on one side covering the user aspect via gathering user feedback and logging user activities, and on the other side determining the accuracy of the designed supporting techniques in terms of standard metrics, such as precision and recall [GK89].

It is important to point out that these steps are tightly connected and have been completed in an iterative way, where the improvements in the description models and the supporting tool lead to adjustments in the Web API analysis and vice versa. The following sections describe each step in more detail.

2.2.1 Analysis of the Current State of Web APIs

Before any progress can me made towards enabling more automated use of Web APIs, we first need to gain a realistic overview of the existing landscape of Web APIs. Given the diversity of the documentation structures and forms, as well as of the underlying technologies, it is important to have in depth knowledge of individual API characteristics, instead of only relying on assumptions. Therefore, the first step towards tackling some of the challenges related to using Web APIs is understanding the shared characteristics and some of the frequently occurring problems. We achieve this by conducting a field analysis in the form of a thorough study of the current state of Web APIs.

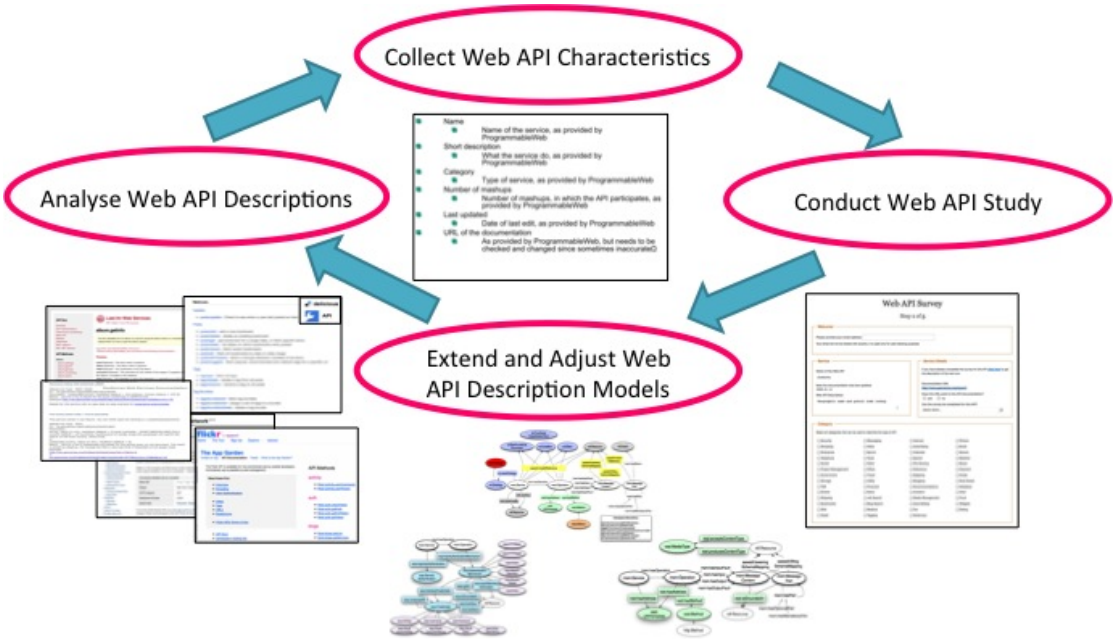


FIGURE 2.2: Analysing Web APIs

The steps followed for performing the analysis of the current state of Web APIs include:

1. **Analyse Web API Descriptions** – Documentations of popular Web APIs are analysed and used to derive a set of common Web API characteristics.

2. **Collect Web API Characteristics** – Determine a set of Web API characteristics, which are used to design a survey for gathering data for each of the identified criteria.

3. **Conduct Web API Study** – The Web API study is carried out.

4. **Extend and Adjust the Web API Description Models** – The results of the study are taken as input for the conceptual solution in terms of Web API description models, identifying points were they can be refined and improved. Finally, the aggregated feedback is used to update the list of covered characteristics.

Figure 2.2 shows the process of conducting the Web API field analysis. We performed two iterations of the process. The first one serves as a reality check and provides valuable details about common description types, characteristics and practices. It also helps to identify deficiencies in existing approaches and gives information about the used technologies and implementation approaches. The second iteration revisits the results of the first one and, in addition, further explores some of the trends and interesting co-relations that were exposed by the initially gathered data.

The objectives of the analysis of the current state of Web APIs are threefold. First, the statistics about the used Web API characteristics, the underlying technologies and followed approaches, should provide a clear overview of the state of APIs on the Web and build a solid foundation for identifying difficulties and challenges. Second, the results serve as a direct input for developing the conceptual solution in terms of Web API description model, since they indirectly pose some requirements through the identification of unwritten best practices and commonly used solutions. Finally, the gathered data indicates, which characteristics are crucial in terms of achieving large coverage of the proposed model, and which are less important.

2.2.2 Describing Web APIs

As already mentioned the solution development is divided into two objectives – conceptual solution and tooling. The conceptual solution focuses on providing the means for formally describing Web APIs. In particular, we advocate the bottom-up, incremental, non-intrusive and modular development of conceptual models that could support the more automated use of Web APIs, focusing in particular on the invocation and authentication tasks.

Bottom-up. First of all, we start with the current state of Web APIs and the available documentation. In particular, the results of the analysis are directly used as input to deriving Web API description models. Therefore, we can be sure that the developed solutions address problems that current developers and Web API users are actually struggling with. In addition, we base our conceptual solution on the existing documentation, instead of creating Web API descriptions directly based on the developed description models.

Incremental and non-intrusive. Second, we aim to enhance existing documentation, instead of enforcing the creation of new descriptions from scratch. We use an approach that is not based on expecting that providers completely replace their current way of documenting APIs, instead focus on providing the means for enhancing text/HTML with rich metadata. In particular, the method followed in defining the Web API description model is based on creating annotations on two different levels – syntactical structuring of the HTML documentation, by identifying the different service elements, and their subsequent enhancing with semantics (see Figure 2.3).

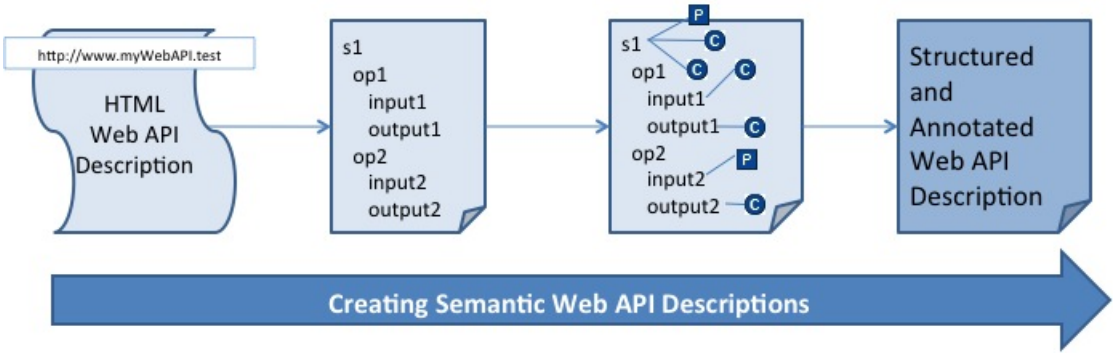


FIGURE 2.3: Creating Semantic Web API Descriptions

Modular. Furthermore, we aim to develop a modular solution towards enabling higher level of automation of service tasks. Instead of defining individual descriptions, which contain, for instance, the relevant details required for performing discovery or composition, the description support is realised by developing a shared core model, based on capturing common service characteristics, and providing extensions to it for enhancing the automation support for individual tasks. Since the scope of this thesis is focused on supporting the usability of Web APIs, by enabling a more automated invocation and authentication process, the extensions are based on capturing API characteristics that are especially targeted towards supporting these two tasks.

We also advocate the adaptation and integration of state of the art solutions for describing and annotating Web APIs. First of all, the development of the description model itself is strongly influenced by previous work in the area, such as MicroWSMO [KV08] and SA-REST [LMR07]. In addition, based on the modularity of the conceptual solution and by defining separate decoupled description parts, the core model can be used independently or in combination with the provided extensions, depending on the current use case. For example, the extensions can easily be applied to enhance either MicroWSMO or SA-REST with the support for the specific task.

In summary, we follow a very traditional process towards developing the Web API description model [SSS06, L99]. The used method is very much aligned with the phases of the ontology engineering lifecycle [Gom98], undergoing specification, conceptualisation, formalisation and implementation. Furthermore, as pointed out by López [L99] the process of defining an ontology is very similar to the software development process, as defined in IEEE 1074-1995 standard [S+97], undergoing requirements analysis, design and implementation.

The developed core service model, as well as the individual task-supporting extensions are evaluated by determining the coverage that they provide in terms of being able to describe currently existing Web API descriptions. In addition, we also determine the level of task support that they offer, focusing on invocation and authentication. Furthermore, the description model is backed up by implementations that support storage of Web API descriptions, search and retrieval, as

well as invocation and authentication. These reference solutions enable the validation of the practical applicability of the designed description model.

2.2.3 Supporting the Creation of Web API Descriptions

In order to ease the use of the introduced description model, the development of supporting tools, that enable the creation and interpretation of semantic Web API descriptions, is crucial. In particular, the implemented tool enables the manual annotation of services, while individual steps such as determining the type of functionality of the API or searching for suitable ontologies are supported by automation components, thus resulting in a semi-automatic annotation solution.

Therefore, the provisioning of support for creating Web API descriptions, with the help of the developed model, is based on the following two steps:

1. Providing a tool with complete support for the manual creation of semantic descriptions, based on existing HTML documentation.
2. Enhancing the tool with automation mechanisms that ease the completion of different annotation tasks.

The annotation tool is evaluated by directly questioning users about its different functions and the effectiveness of the support that they provide. This is complemented by gathering feedback from participants in training and hands-on sessions, where the tool is used. Finally, comments and input from a number of use cases are also used to evaluate the different features of the tool. Based on this data, conclusions can be drawn about the effectiveness of the tool support in comparison to having to create the semantic descriptions without it, by using a text editor, for example.

The following part discusses in detail related work and sets the context of our research.

Part II

Context and Related Work

Chapter 3

Web Services and Web APIs

While research in the area of Web APIs is still relatively new and not as prolific in terms of approaches and implemented systems, there is quite a plenitude of work on Web Services and Semantic Web Services. This chapter provides an overview of existing description frameworks, suitable application use cases and addressed challenges, in order to provide a foundation for determining the potential for reusing some of the previous work but also to identify open issues and remaining difficulties.

The following sections describe research related to Web services and Web APIs, including underlying principles and technologies, description formats, use of semantics and overall integration approaches. We divide the state of the art in three main parts. We start by providing an overview of the fundamentals related to Web services and Web APIs, and reflect on the different ways of capturing service properties in terms of documentation forms and formats (Sections 3.1, 3.2 and 3.3). Following is a section on semantic approaches developed in the context of Web services and Web APIs (Section 3.4). Finally, we shortly provide an introduction to Linked Data principles and work targeted at supporting the integration of data and services on the Web (Section 3.5).

The work presented in this thesis aims to address some of the challenges faced by Web APIs and to support their more automated use, in order to contribute to a dynamic and integrated Web, where Web APIs, Web services and Linked Data sources can be employed together, so that users and applications can access data without differentiating whether its source is a website, a Web API or Linked Data. Therefore, we address precisely these three areas of related work because in the context of developing a solution that contributes towards Open Services on the Web, it is important to be aware of:

- The basic underlying technologies, which define the framework for the possible solutions;

- The existing solutions for capturing service properties, in terms of descriptions (this covers the structuring of service details on a syntactic level);
- The use of semantics for enhancing services and the higher-level of automation support for common service tasks, provided by employing semantics;
- The approaches that already contribute towards the integrated used of services and data on the Web.

For each of theses lines of work we analyse their applicability in the context of using Web APIs and discuss their features and limitations. In particular, while describing the solutions for capturing service properties, we determine the:

- The service characteristics that can be captured;
- The support that they provide for common service tasks such as discovery and composition, focusing mainly on invocation and authentication;
- The complexity of the description formalism and the format itself.

Related work in the area of enhancing services with semantics is analysed in terms of:

- The type of semantic annotations;
- The definition of a service description model;
- The complexity of the description formalism and the format itself.

As a result we are able to gain a clear overview of the state of the art so far, identify approaches and solutions that can be adopted and determine existing limitations, which need to be addressed.

3.1 Web Services

"Classical" Web services, based on WSDL [W3C07a] and SOAP [W3C07b], have for a long time dominated the service world, playing a major role in the interoperability within and among enterprises and serving as the basic construct for the rapid development of low-cost and easy-to-compose distributed applications in heterogeneous environments [PTDL08]. Web Service [ACKM04] technology enables publishing and consuming functionality of existing applications, facilitating the development of systems based on decoupled and distributed components. Therefore, Web services can be seen as reusable building blocks for creating distributed systems,

which allow companies and individuals to make their digital assets available to the global community in a simple and effective manner.

Web services, or more precisely services based on the Web services technology stack [ACKM04], are an established technology and are commonly used. However, despite their popularity there are still **only a few services available** (28 000[1] in comparison to 112 million of registered domains[2] and over 182 million websites[3]) and this number is not significantly increasing[4]. Moreover, services based on WS- * (SOAP, WSDL, WS-Addressing, WS-Messaging and WS-Security, etc.) [ACKM04] are argued to have **high complexity**, require trained developers, relatively sophisticated infrastructure or extensive tooling [PZL08]. This is especially true if traditional WS are directly compared to Web APIs that rely only on HTTP for message transfer and URIs for endpoint and resource identification (see Section 3.2).

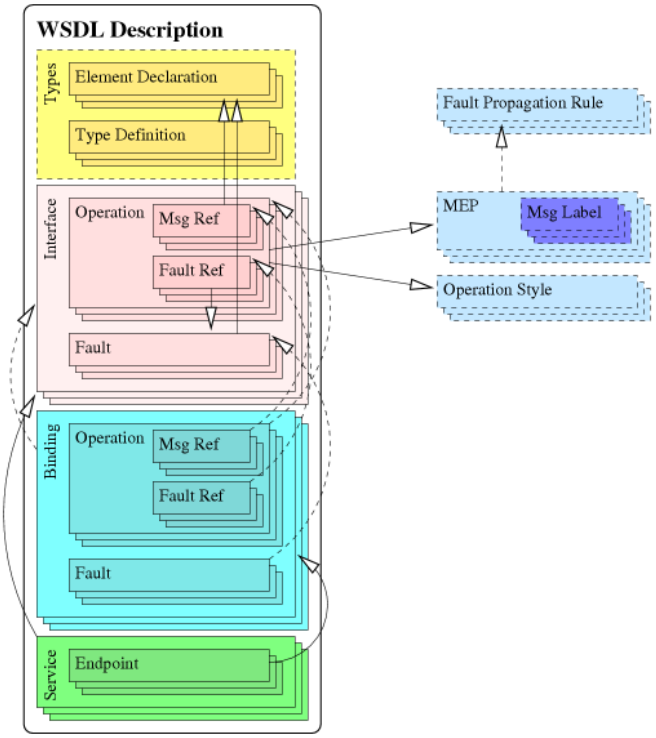


FIGURE 3.1: WSDL Structure

The Web Service Description Language (WSDL) [W3C07a, W3C01] is used to describe the service access endpoint, providing a machine-processable description of the structure of the service, its operations and the request and response messages. It consists of two main parts, including an

[1]Source: `http://webservices.seekda.com/`

[2]Daily updated statistics at `http://www.domaintools.com/internet-statistics/`.

[3]Source: Netcraft `http://news.netcraft.com/archives/2008/10/29/october_2008_web_server_survey.html`

[4]No significant increase in number during the past three years, source: `http://webservices.seekda.com/`

abstract and a concrete definition (see Figure 3.1[5]). The abstract part includes an interface that is a group of operations associated with the corresponding messages that can be sent and received. However, none of these elements is restricted to a specific data format or transport protocol. As a result the modular and implementation independent approach towards describing the service promotes reusability of the individual parts and separation of design and programming concerns. The abstract section of the WSDL file is complemented by concrete implementations that give the actual transport and wire formats as well as the network address for the endpoint definition. More importantly, WSDL is XML-based and its file structure, as well as all possible information items, is defined in the `http://www.w3.org/2007/06/wsdl/wsdl20.xsd` XML schema. Therefore, WSDL-based service descriptions are not meant for human interpretation but rather serve as the basis for completing common tasks, such as discovery and invocation, through automated machine processing. The structure of WSDL is visualised in Figure 3.1.

In summary, WSDL specifies: 1) the supported operations for invoking the Web service; 2) its transport protocol bindings; 3) the message exchange format; and 4) its physical location. In this way, the WSDL description contains all information necessary for invoking a service. A WSDL service definition is commonly used with SOAP [W3C07b], which specifies the messages exchanged between the service consumer and provider. A SOAP message consists of an *Envelope*, which groups the *Header*, containing communication specific details, and the *Body*, including the actual message. SOAP is XML-based as well and a sample SOAP message can be seen in Listing 3.1[6].

```
1  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
2   <env:Header>
3    <n: alertcontrol  xmlns:n="http :// example.org/alertcontrol">
4     <n: priority >1</n: priority >
5     <n:expires>2012-06-22T14:00:00-05:00</n:expires>
6    </n: alertcontrol >
7   </env:Header>
8   <env:Body>
9    <m:alert xmlns:m="http://example.org/alert">
10    <m:msg>Project technical call at 3pm</m:msg>
11   </m:alert>
12  </env:Body>
13 </env:Envelope>
```

LISTING 3.1: Sample SOAP Message

WSDL and SOAP can be used in conjunction with WS-Addressing, WS-Messaging, WS-Security, etc., specifications, which provide further support for particular use cases. As already stated in the introduction of this chapter, we discuss the features of the individual approaches for capturing service properties in order to be able to make a comparison and draw conclusions. In the case of Web services described in WSDL, the following service properties are specified – service, operation, input, output, types of the inputs and outputs and implementation binding.

[5]Image source – D11v0.2 WSMO-Lite [KV07], `http://http://www.wsmo.org/TR/d11/v0.2/20070421/`

[6]Source – SOAP 1.2, `http://www.w3.org/TR/soap12-part1/`

In terms of the support that is provided for common tasks, based on the WSDL description, invocation can be handled directly, without extensive manual effort (see Chapter 4). In contrast, regarding discovery and composition, developers need to manually search for services in repositories such as UDDI [CHvRR04], in order to subsequently design service compositions and develop software that is able to invoke and manipulate them. Authentication information is not directly described in the WSDL file but can be handled by adding WS-Security on top of the SOAP messages.

Finally, in terms of complexity, a WSDL file is relatively simple, since it does not contain very many elements. WSDL is written in XML, which is not meant for human interpretation and might, therefore, require additional documentation in a textual form. However, it has the benefit of being machine processable and by using an XML schema, it can be guaranteed that the file is well-formed. A variety of XML editors can be used to ease the work with WSDL, visualising the structure or summarising the main elements.

3.2 RESTful Services and Web APIs

Currently, there is an increasing importance and use of Web applications and APIs and sometimes it is even argued that they have displaced Web services [Rod08]. This trend is strongly supported by the growing popularity and use of Web 2.0 technologies, because many Web 2.0 applications offer Web APIs as access points for interaction with application resources[7]. Moreover, Web APIs enable combining heterogeneous data coming from diverse sources, in order to create data-oriented compositions called mashups [Wor07].

Web APIs, which conform to the REST principles [Fie00] are commonly referred to as RESTful services [RR07]. The representational state transfer (REST) paradigm was first described by Roy Fielding in his Ph.D. dissertation [Fie00] denoting an architecture style of large-scale distributed systems. A RESTful service is a simple service based on REST principles and frequently implemented by using HTTP. RESTful services comprise a collection of resources and comply with the following principles[8]:

- *Client-server* – there is a clear separation between the client and the server, which communicate via a uniform interface. This separation is realised by the client not being concerned with server-specific tasks, such as data storage, and the server not being concerned with client-specific tasks, such as user state-information.

[7] Twitter API Traffic is 10x Twitter's Site, `http://blog.programmableweb.com/2007/09/10/twitter-api-traffic-is-10x-twitters-site/`

[8] A complete and exhaustive list is available in [Fie00].

- *Stateless* – The client-server communication is conducted in a stateless manner, i.e., there is no client context being stored on the server between requests.
- *Cacheable* – Clients should be able to cache responses. This requires that the responses contain caching information.
- *Uniform interface* – The communication between the client and the server takes place via a fixed interface, which enables their decoupled and independent development. This interface should be kept simple and is shared among all APIs. For instance, this can be achieved by using the uniform methods of HTTP protocol.
- *Layered system* – The client is not aware of whether it is connected directly to the server or not. The involvement of intermediary servers should be transparent to the client.
- *Code on demand* – Adding executable code can be used to temporally extend the functionality of the client. For example, a client can download and use a JavaScript or java applet to add further functionality, such as encryption of the communication. Still, the server is not aware of any encryption routines / keys used in this process.

There are also a number of guidelines that should be followed while defining the interface. These are especially relevant in the context of RESTful services, since they specify how the endpoint should be defined and how the communication is realised.

- *Uniquely identified resources*: All resources are uniquely identified by URIs [BLFM05], which enable the addressing and retrieving of resources on a global scale. In the context of services, a RESTful service exposes a set of resources, which identify the endpoints for interaction with clients. The resources themselves are conceptually separate from the representations that are returned to the client. For example, an article about Milton Keynes can be available as HTML, XML and also in different languages.
- *Manipulation of resources through representations*: Every interaction with a resource is realised through its representation, which is available to the client. All resources can be manipulated by using a fixed set of operations; in the case of an HTTP implementation, theses would be – GET, POST, PUT, and DELETE. GET is used to retrieve a resource representation, while PUT replaces an existing resource or creates it, if it does not exist. POST creates a resource and returns its URI, and DELETE removes the resource.
- *Self-descriptive messages*: Each message includes the necessary information about how to process it.
- *Hypermedia as the engine of application state (HATEOAS)*: The state of the client can change only through actions that are dynamically identified within hypermedia (for example, through links in text) by the server.

In summary, the essence behind RESTful services is that the interfaces are defined in terms of resources, whose representations can be retrieved and manipulated via a fixed set of operations. The REST principles have been developed in order to ensure a number of architecture characteristics, which are beneficial for the developed systems. These include, independent deployment of components, scalability of the component interaction, uniform interfaces, responsibilities decoupling, etc. When applied to the Web and the development of Web applications, REST helps to ensure Web conformity and interoperability.

It is important to point out that currently, most available Web APIs do not follow all of these REST principles [MPD10a] (see Chapter 6). In fact about two thirds of the Web APIs do not define the interface in terms of resources manipulated via the HTTP methods but rather in terms of parameterised arbitrary operations (see Chapter 6). Therefore, when we talk about Web APIs and develop an approach, which aims to support their automated use, we cannot automatically assume that the REST principles apply. As we will see in the chapter presenting the results of the two Web API surveys (see Chapter 6), this is actually not the case. Therefore, RESTful services represent a subset of all Web APIs. To this purpose we introduce a definition for Web APIs, which is used throughout this thesis (see Chapter 7). In essence, a Web API, as defined within the scope of this work, is an endpoint[9] that provides access to functionalities or resources in a programmable (i.e. machine-oriented) way over the World Wide Web, via Web-related standards such as URIs and HTTP. This definition is described in more detail in Chapter 7.

Overall, Web APIs and WSDL-based services are suitable for different use cases. Web APIs fit well in the context of developing applications with a Web setting, especially since the underlying technologies are aligned with the fundamental workings of the Web. In contrast, traditional Web services are suitable for an enterprise environment, where they enable the machine-to-machine communication within a system or between decoupled systems [PZL08]. Furthermore, they are more suited for scenarios, which require security measures or transactions. Therefore, Web APIs have a number of advantages, especially for use cases based on Web technologies. They are simpler to use because they do not need implementations of the WS technology stack and instead rely directly on HTTP for message exchange and URIs for endpoint definition.

3.3 Description Models for Web APIs

Web services and Web APIs enable publishing and consuming functionalities and resources in a programmable way, thus facilitating the development of applications based on combining and pipelining individual inputs and outputs. As already mentioned, WSDL is an XML-based language for describing the interface of a Web service, in a way that supports automated machine

[9] An *endpoint* in this context is a specific location for accessing a service, specified via URI, using a specific protocol and data format.

processing. In this section we discuss the existing description possibilities for capturing Web API characteristics. Currently the majority of the Web APIs have only textual HTML-based documentation and do not follow any particular guidelines prescribing the used format or the included details. It is up to the providers to decide, which information about the API is made available and how to structure it. The documentation is given in textual form, sometimes on a single webpage, sometimes as part of a collection of interlinked pages. There is no clear motivation as to why this is the case but the lack of a widely accepted standard format, as for example XML, and a standardised structure, as for example WSDL, are characteristic for Web APIs and lead to a number of challenges.

In particular, descriptions that require human interpretation cannot serve as the basis for scalable and interoperable solutions. Currently, developers need to manually search for suitable APIs, read through the documentation, identify the service properties, browse through discussion groups and forums and implement custom solutions, a lot of the time through trial-and-error testing. This approach for using Web APIs is not scalable, the individual solutions are not interoperable, changes in the documentation cannot be directly propagated in the implementation, etc.

A unified view on Web APIs, through a common description language alleviates these difficulties. Furthermore, as argued by Renzel et al. [RSK12] formal descriptions, especially in the context of RESTful services, but also for services in general, allow for easier automated service access and are often used for automated service composition [RS04, Pau09]. In this context, we distinguish between approaches that require the creation of new descriptions from scratch (e.g. WSDL 2.0 and WADL [Had06]) and approaches that rely on enhancing existing HTML documentation (e.g. hRESTS [KGV08]). We discuss these in the following paragraphs.

It is important to point out that if all Web APIs were RESTful, the situation would be a bit different. The REST principles fix the way of conducting the communication between the client and the server, there is a known set of operations to use on the resources, the way of defining the URIs is also implicitly specified, and the output format is set via content negotiation and not by using parameters. In summary, a lot of the service properties that would need to be specified in a description, are already determined because of the architectural decisions. Ideally, through the use of hyperlinks, link following and request construction, based on the hypertext and client state, client development can be done without explicitly documenting Web APIs. However, what would still be necessary is a list of all the resources. This is sometimes addressed by implementations where the calling of the root URI, returns a list of all available resources. Still, the current sate is that since there is no standard language that a machine could interpret, providers commonly document the URIs, the used HTTP methods, and structures of representations (e.g. as XML and JSON) directly in HTML as part of webpages. Furthermore, in the context of providing a bottom-up solution and using semantics for enabling a higher level of automation

to common service tasks (Section 3.4), an existing description that includes service properties, which can be annotated, is a prerequisite. In summary, as already pointed out while describing our approach, we aim to provide a real world, incremental solution covering a large percentage of the existing Web APIs. Currently, the majority of the Web APIs have textual documentation[10] and only about one third of the APIs are RESTful [MPD10a], therefore, we adopt a description-based view on Web APIs.

WSDL. Similarly, to Web services, which provide access to the functionality of existing components, Web APIs and Web applications conforming to the REST paradigm, provide access to resources by using the WWW as an infrastructure platform. Based on this parallel between the two types of services, WSDL was extended to Version 2.0 [W3C07a] for describing RESTful services as well. As a result WSDL is a platform- and language-independent form of describing both Web services and RESTful services on a syntactic level.

Still, WSDL is not particularly suitable for describing Web APIs that conform to REST. It is not especially designed for resource-oriented services and as a result, everything has to be described in an operation-based manner. In addition, WSDL lacks support for including simple links, for instance, there is no mechanism to describe new resources that are identified by links to other documents, which is essential for RESTful services. However, we aim to provide coverage for all APIs and, as already pointed out, only a portion of the Web APIs is RESTful.

In section 3.1 we already analysed the service properties that can be captured by WSDL, the provided task support, and the level of complexity of both the format and the formalism. Therefore, here we focus only on its suitability to describing Web APIs. Given the service properties that it can describe and the fact that there are already a lot of solutions based on WSDL, this Interface Description Language (IDL) is in general suitable for APIs. It would be necessary to define a mapping for the resource-oriented Web APIs to operations, by combining the resource and the HTTP method in order to define the corresponding operation. Furthermore, the existing textual documentation would have to be transferred into WSDL. Finally, developers would need to learn to deal with WSDL instead of simply reading a natural language description. Therefore, we do not exclude WSDL as a possible description format.

WADL. In contrast to WSDL, the Web Application Description Language (WADL) [Had06] was especially designed for describing RESTful services in a machine-processable way. It is also XML-based and is platform and language independent. As opposed to WSDL, WADL models the resources provided by a service, and the relationships between them in the form of links. A service is described using a set of resource elements. Each resource contains descriptions of the inputs and method elements, including the request and response for the resource. In particular, the request element specifies how to represent the input, what types are required and any specific

[10] While conducting the two Web API surveys we did not find any APIs without documentation.

HTTP headers. The response describes the representation of the service's response, as well as any fault information, to deal with errors.

In the case of Web API descriptions in WADL, the following service properties are specified – resource, method and resource type, where a resource has parameters (in the request) and representation (in the response). The resource URIs are defined starting from the resource base, relative to each other, depending on the node nesting. In terms of the support that is provided for common tasks, WADL is very similar to WSDL – based on the WADL description, invocation can be handled directly (see Chapter 4). Regarding discovery, there is no popular registry, such as UDDI, for publishing the descriptions, however, composition support based solely on the description formalism is better because the links between the resources are explicitly defined. Authentication information is not directly described in WADL and has to be handled by additional protocols on top, similarly to WS-Security for SOAP messages.

In terms of complexity, WADL and WSDL are very similar and are relatively simple, not containing many elements or any complex nesting structures. WADL files are written in XML and benefit from being machine processable, even though, that might require supplementary documentation in a textual form. Currently, neither WADL nor WSDL are widely accepted and used for Web APIs and RESTful services [All11]. Instead the documentation is usually given in natural language as part of a webpage (see Chapter 6). It is up to the developer to decide what structure to use and what information to provide. However, plain text/HTML descriptions do not support the automated interpretation of the service properties, which means that if a developer wants to use a particular service, he/she has to go to an existing description webpage, study it and write the application manually. Therefore, current research proposes the creation of machine-interpretable descriptions on top of existing HTML documentation by using **microformats** [KC06]. Microformats offer means for annotating human-oriented webpages in order to make key information automatically recognisable and processable.

hRESTS. One particular approach for creating machine-processable descriptions for RESTful services by using microformats is hRESTS (HTML for RESTful Services) [KGV08]. hRESTS enables the marking of service properties including operations, inputs and outputs, HTTP methods and labels, by inserting elements and attribute-value pairs within the HTML. In this way, the user does not see any changes in the webpage, unless the used stylesheet highlights these tags. However, based on these annotations, the service can be automatically recognised by crawlers and the service properties can directly be extracted by applying a simple transformation. The HTML documentation, with the tags marking the service properties, contains the syntactical information of the described Web API and therefore, no longer relies solely on human interpretation.

hRESTS specifies only a very limited number of properties – service, operation, input, output, HTTP method and labels. Therefore, this approach is very simple and with low complexity

in terms of the description formalism and the format - HTML tags. However, its simplicity exhibits some limitations, since there is no support for describing parts of the input or the output, the endpoint of the API cannot be captured, and it cannot be described what the input and output types are. This affects also the support that hRESTS provides for common service tasks. Basic Web API discovery can be done automatically by recognising a webpage as a Web API description and being able to identify the annotated service properties. However, there is no support for composition and invocation still has to be implemented through realising a custom client, especially since some key information such as the endpoint or how the input is transmitted (as part of the HTTP body or as URI parameter) is missing (see Chapter 4). Authentication information also cannot be described with the help of hRESTS, so it has to be handled as part of the client application solution.

It should also be pointed out that hRESTS is not directly suitable for describing resource-oriented APIs, but instead takes an operation-centric view. This can be solved by creating derived operations by combining the used HTTP method and the resource [KVPM11] (see Chapter 7). However, approaches, based on making existing Web API service descriptions machine-processable by using HTML tags are simpler than creating descriptions from scratch and more lightweight, since they involve only 5-6 tags, as opposed to WSDL and WADL. In addition, as already mentioned, they can be applied directly on already available documentation, rather then creating new service description files from scratch, as in the case with WSDL and WADL. The adoption by developers is also easier, since the creation of a machine-processable Web API service description is equivalent to Web content creation or modification.

3.4 Semantic Descriptions of Web Services and Web APIs

This section focuses on research work related to the semantic description of Web services, exploring the potential for reusing and adopting the richness of approaches developed in this area.

Machine-processable service descriptions, such as XML-based files, support the creation, publication and consumption of services. However, the service descriptions remain on a syntactic level, capturing the individual properties but containing no information about the actual meaning of the performed operations or their inputs and outputs. As a result, finding and composing services still requires manual effort [SWGS05]. In order to address this challenge, research on **Semantic Web Services** (SWS) has been devoted to reducing the manual effort required for manipulating Web services [FLP+06, MSZ01]. The main idea behind this research is that tasks such as the discovery, negotiation, composition and invocation of Web services can have a higher level of automation, when services are enhanced with semantic descriptions of their properties

(see Figure 3.2[11]). In particular, SWS apply inference-based techniques on formalised semantic descriptions, in order to better support the analysis of Web services.

Semantics are specified through **ontologies**, which are the means to avoid the misinterpretation of terms by formally describing a conceptualisation [Gru93]. Ontologies are formalised by using a logical language, which enables machines to process the encoded knowledge and to derive additional facts. There are different types of ontologies; some can have a simple form of a taxonomy tree that relates terms by specialisation and generalisation relations, while others may use complex logical expressions to describe terms in relation to each other. OWL [BvHH+04] defines a formal language for defining ontologies. It is a W3C recommendation for describing ontologies based on other W3C recommendations, RDF [Hay04] and XML. RDF is used for modelling metadata and is a general method for conceptual description of information, based upon the idea of making statements about resources in the form of subject-predicate-object expressions (triples).

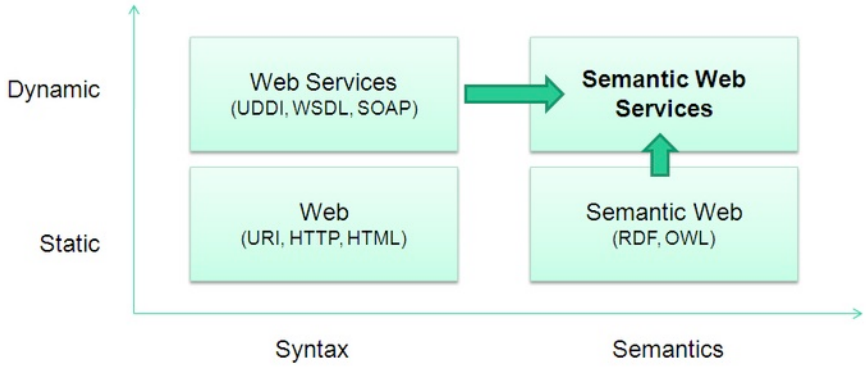


FIGURE 3.2: Combining SOA and the Semantic Web

Most SWS technologies use ontologies for capturing service-specific data models, which serve as a basis for providing task automation support but also for tackling the interoperability problem at the semantic level. These models provide concepts to populate the generic description service template, which captures the main aspects describing the essence of a Web service, with specific entities (e.g. Flight, Hotel) and functionalities (e.g. bookTrip). On the other hand, the actual Web service characteristics, that define what a service is and what it looks like, can also be captured in a service model ontology. Both the data model ontologies and the service model ontologies are collectively referred to as **Web service ontologies** [SWGS05].

Since Web APIs are currently facing similar challenges as WSDL-based Web services, and require even further manual effort due to the lack of machine-interpretable documentation, such as an XML description file, research related to creating and processing SWS descriptions should be carefully considered as a basis for semantically enriching Web APIs. In addition, if Web APIs can be semantically described by directly applying or by adapting existing SWS formalisms,

[11] Source [DM08]

this would support the reuse of already developed semantic-enabled discovery, composition and invocation approaches.

Web Service Semantics. Generally four main types of semantics have been defined, that can be included in order to enhance a service description [SVSM03, She03]. Each type of semantics relates to a particular set of service features. In particular, these are:

1. *Informational semantics*, also referred to as data semantics, which relate to the data processed by the service. These are usually expressed through mappings to ontology elements.
2. *Functional semantics*, which relate to the functionality of the service.
3. *Non-Functional semantics*, which capture non-functional characteristics such as Quality of Service (QoS), pricing, security or reliability.
4. *Execution semantics*, which pertain to the executional behaviour of the service.

All SWS approaches cover these four types of semantics, or at least a subset of them, thus enabling the automated processing and reasoning over certain service features. Naturally, there is a trade-off between the expressivity of the description models and the complexity that has to be taken into consideration.

Semantic Web Service approaches can be grouped into two main types, based on the two main trends in the area. There are *top-down* approaches, which are driven by high-level views over Web services and require the introduction of new models and description forms, and *bottom-up* approaches, which rely on enhancing and enriching existing Web services technologies, instead of suggesting a completely new solution. OWL-S [Mar04] and WSMO [RKL$^+$05] are two top-down approaches. In the following sections we also discuss some commonly used bottom-up approaches (SAWSDL [FL07] and WSMO-Lite [VKVF08]).

DAML-S and OWL-S. DAML-S [ABH$^+$01] represents an initial effort towards enabling the automatic location of Web Services on the basis of their capabilities, which neither SOAP nor WSDL are of any help for. Therefore, it covers the abstract description of the capabilities of the service as well as the specification of the service interaction protocol, to the actual messages that it exchanges with other Web Services. In particular, the *Service Profile* is used to describe the functionalities that a Web Service wants to provide. For instance, based on DAML-S Paolucci et al. [PKPS02] propose an algorithm for matching service advertisements and service requests, with different degree of match depending on how well the request and the advertisement capabilities overlap.

DAML-S evolved into OWL-S [Mar04], which defines an upper ontology for semantically describing Web services. OWL-S enables different levels of expressivity and decidability, depending on what subtype of the Web Ontology Language (OWL) [BvHH+04] is used – *OWL-Lite*, *OWL-DL* or *OWL-Full*. The high-level objective of OWL-S is to [MBM+07]: (1) provide a general purpose Semantic Web service framework, (2) support automated service usage and management by software agents, (3) build on existing Web and Semantic Web standards, (4) support a complete service lifecycle.

As visualised in Figure 3.3[12], OWL-S consist of three main parts: 1) the *Service Profile* for advertising and discovering services, which contains the name of the Web service, its provider, a natural language description, and a formal functional description that is defined in terms of inputs, outputs, preconditions and effects; 2) the *Service Model*, which gives a detailed description of a service's operation and describes how the Web service works; and 3) the *Service Grounding*, which provides details on how to access the service and how to interoperate with a service, via messages.

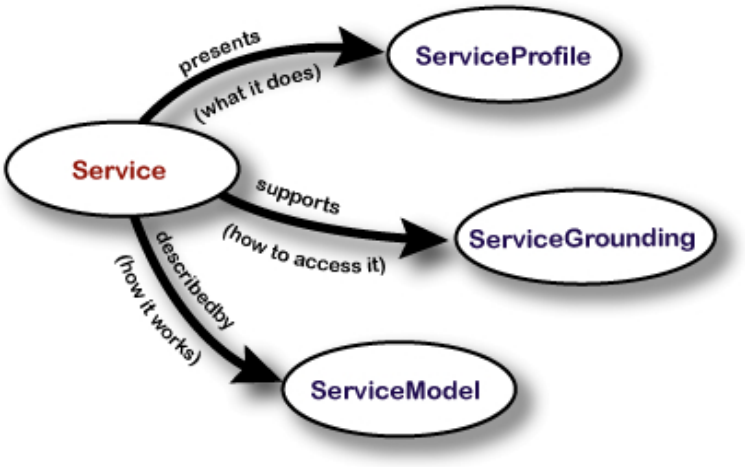


FIGURE 3.3: OWL-S Main Concepts

The service profile supports manual service discovery, based on the natural language descriptions, as well as automatic discovery, based on the formal functional description. The service model provides information for determining whether the communication between a client and the Web service can be carried out successfully. Finally, the service grounding defines the mapping between the semantic descriptions and technological implementation in order to conduct the actual message exchange. One weakness of OWL-S is related to conceptual insufficiencies, in particular, addressing the inadequacy of the process description language [LRPF04]. Related to Web APIs, OWL-S provides a good basis for identifying the semantic information, which needs to be included in the service description, in order to effectively support the discovery, composition and invocation tasks. However, it needs to be investigated whether the level

[12]Source – OWL-S: Semantic Markup for Web Services, `http://www.w3.org/Submission/OWL-S/`

of complexity that OWL-S introduces makes it unsuitable for solutions in the context of Web applications and APIs.

WSMO. Another SWS description formalism is the Web Service Modeling Ontology (WSMO) [RKL+05]. It defines four top-level notions including: 1) *Ontologies* that define formalised domain knowledge; 2) *Goals* that describe the client's objective in using the Web service; 3) semantic description of *Web Services*; and 4) *Mediators* for enabling interoperability and handling heterogeneity. In contrast to other description formalisms, WSMO proposes a **goal-based** approach for Semantic Web Services. A client formulates requests in terms of goals, which formally describe the objective to be achieved, while abstracting from the actual technical implementation. The task of the system is to automatically discover and execute Web services, which satisfy the client's goal. In this way, Web services are not described only in terms of their capabilities but also in terms of the problems that they solve. This approach is supported by the integrated mediators, which enable the handling of potentially occurring heterogeneities. Figure 3.4[13] visualises the four main elements of WSMO.

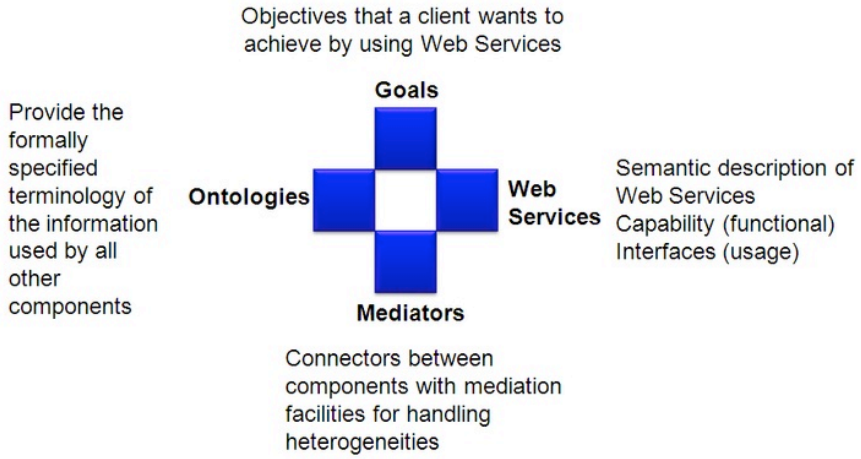


FIGURE 3.4: The Top-level Elements of WSMO

WSMO features are claimed to be more advanced than OWL-S [LRPF04], including options for detailed specification of non-functional properties, mediation for handling heterogeneity, and description of choreography semantics. WSMO promotes a goal-driven approach, which goes beyond the idea of merely annotating Web services, aiming to provide means of expressing service offers and needs. In addition, WSMO defines its own specification language WSML [dBLK+05] that covers all ontology languages that are considered for the Semantic Web, and provides reasoners for them, making it therefore widely applicable. Moreover, there are implementations of execution environments for Semantic Web Services, namely WSMX as the WSMO reference implementation (see www.wsmx.org) [FKZ08] and IRS as a goal-based broker for Semantic Web Services [DCG+08].

[13] Source – Web Service Modeling Ontology (WSMO), `http://www.w3.org/Submission/WSMO/`

Similarly to OWL-S, WSMO provides details about the semantic information necessary for creating semantic descriptions of Web APIs, which support the automation of service tasks. In addition, WSMO provides a new perspective to services by introducing the concept of goals. Still, it is important to point out that both OWL-S and WSMO bring a certain level of complexity, which requires developers with good ontological knowledge and experience in using the formalisms. This somehow contradicts the current trends in Web API development, where the majority of the descriptions are given in simple HTML and the technology is more accessible to less-trained developers.

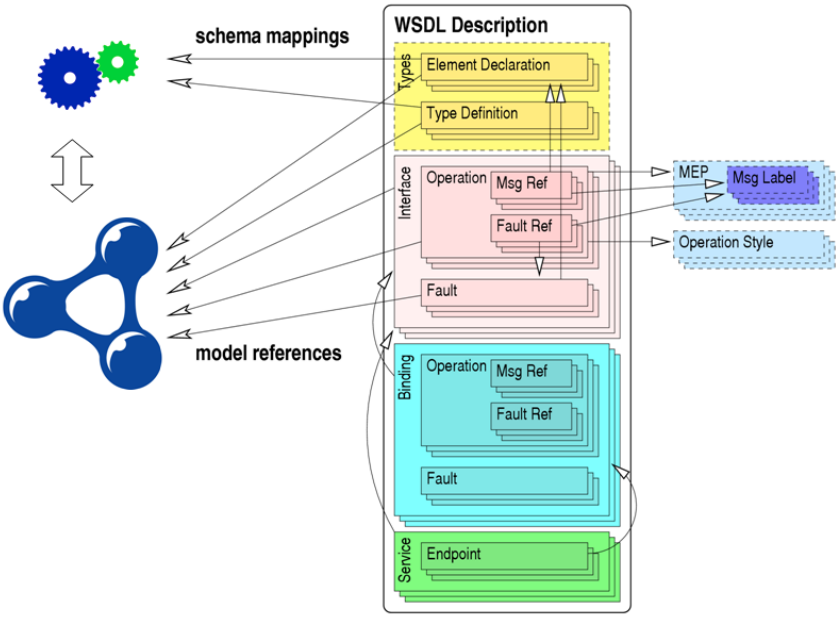


FIGURE 3.5: SAWSDL Elements and Their Relationship to WSDL

SAWSDL. A more lightweight description formalism is the Semantic Annotations for WSDL and XML Schema (SAWSDL) [FL07], which currently is the only official W3C recommendation for Semantic Web Services. It defines a set of extension attributes for WSDL, which enable the linking of semantic entities to the service description. Many of the concepts in SAWSDL are based on an earlier effort WSDL-S [AFM+05]. SAWSDL consists of two parts: 1) Mappings of XML schema definitions to ontology concepts, which specify the translation of data between the semantic layer and the layer of technical implementation. In particular, *liftingSchemaMappings* capture how to lift data represented in XML to its semantic counterpart. Conversely, *loweringSchemaMappings* capture how to carry out the inverse process, which is to transform data represented semantically into the XML representation the service expects. 2) Semantic annotation of WSDL elements through references to ontology concepts done with the help of *modelReference*. Figure 3.5[14] visualises the two main SAWSDL elements and how they relate to the individual WSDL description elements.

[14]Source – Semantic Annotations for WSDL and XML Schema, `http://www.w3.org/TR/sawsdl/`

It is important to point out that SAWSDL does not actually specify a language or a model for describing Semantic Web Services. In essence, what SAWSDL provides is the means to link service properties to semantic annotations, as exemplified in Listing 3.2. It is a simple but powerful method for enhancing WSDL descriptions with metadata.

```
1 <wsdl:operation name="CheckAvailabilityRequestOperation"
2   sawsdl:modelReference="http://www.exmaples.org/taxonomy/Categorization#Electronics"
3   ...
4 </wsdl:operation>
```

LISTING 3.2: Example of using SAWSDL for annotating a WSDL operation

SAWSDL defines a set of minimal extensions that allow to define the semantics of a service or its parts, by pointing to concepts from externally defined semantic models. Even though the resulting annotations are not as expressive as the OWL-S or WSMO ones and adding of semantic information is limited to service properties associated with semantic concepts, several recent works build on this lightweight SWS approach [VKVF08, KV08]. In the context of annotating Web APIs, given that the service properties are identified within the HTML documentation, the SAWSDL approach is very promising because it enables adding semantic information directly on top, by linking service keywords to ontology elements.

WSMO-Lite. SAWSDL supports the development of lightweight service description formats, in order to enable the creation of Semantic Web Service descriptions. However, SAWSDL only provides the solution to how to add annotations, it does not specify the type and meaning of the made annotations. WSMO-Lite [VKVF08] supports the lightweight descriptions of WSDL services, by filling the SAWSDL annotations with concrete service semantics. WSMO-Lite defines four aspects of service semantics including: 1) *Information Model*, which defines the data model for input, output and fault messages. This type of annotations are visualised through the *A1* and *A2* arrows in Figure 3.6[15]; 2) *Functional Semantics*, which define the functionality, which the service offers to a client when it is invoked. This type of annotations are visualised by the *A3* and *A4* arrows; 3) *Behavioural Semantics*, which define details specific to a service provider, or the service implementation or its running environment; and 4) *Nonfunctional Descriptions*, which define nonfunctional service properties such as quality of service or price (*A5*). These service semantics are linked to a WSDL description by applying the SAWSDL annotation approach.

In particular, WSMO-Lite adopts the types of annotations provided by SAWSDL and, therefore, supports two types of annotations – *reference annotations* and *transformation annotations*. A reference annotation points from a WSDL component (XML Schema message or type definition, interface, operation, binding, service, or endpoint) to a WSMO-Lite semantic concept. A transformation annotation specifies a data transformation in terms of liftings and lowerings. A major

[15] Source – WSMO-Lite: Lightweight Semantic Descriptions for Services on the Web, `http://www.w3.org/Submission/WSMO-Lite/`

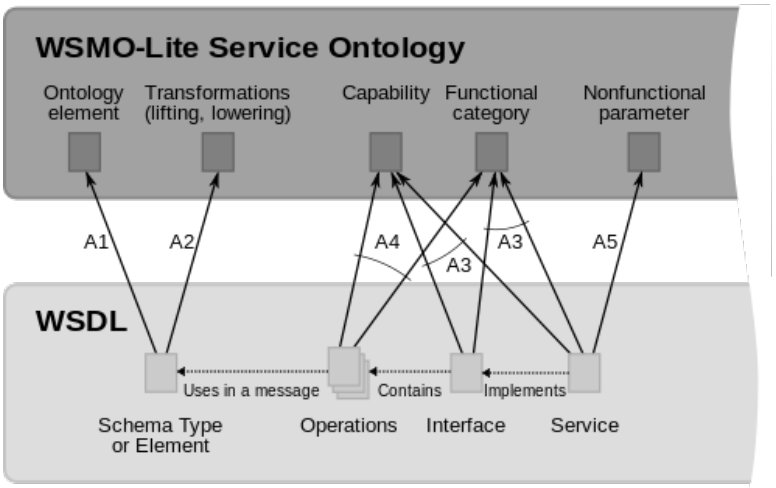


FIGURE 3.6: WSMO-Lite for Annotating WSDL

advantage of the WSMO-Lite approach is that it provides means for describing service semantics without really being bound to a particular service description format (for example, WSDL). As a result, service properties, for example, marked with hRESTS, in an HTML documentation of a RESTful service can be used as annotation points for linking WSMO-Lite semantic concepts in the same way as using service properties described in WSDL. In essence, WSMO-Lite identifies the main kinds of semantics that can be provided for services and does not define a service model by itself.

Approach	Functional	Non-Functional	Informational	Behavioural	Processes
DAML-S / OWL-S	yes	yes	yes	yes	yes
Capabilities Model	yes	no	yes	yes	no
$^{my}Grid$ & SADI	yes	yes	yes	no	no
ASWS	yes	no	yes	yes	no
DIANE	yes	yes	yes	yes	no
Semantic MOBY	no	yes	yes	no	no
SWSO	yes	yes	yes	yes	yes
WSDL-S	yes	no	yes	yes	no
WSMO	yes	yes	yes	yes	yes
COWS	yes	yes	yes	yes	yes
QuASAR / ISPIDER	yes	no	yes	no	yes
Web service ontology	yes	no	yes	yes	yes
BPEL4SWS	yes	yes	yes	no	yes
SAWSDL	not explicitly	not explicitly	yes	no	no
FUSION Ontology	yes	yes	yes	no	no
YASA	via SAWSDL	via SAWSDL	via SAWSDL	via SAWSDL	no
MSM	via WSMO-Lite	via WSMO-Lite	yes	no	no
ER Model	yes	yes	yes	yes	yes

TABLE 3.1: Semantic Web Service Approaches

Further Semantic Web Service Approaches. Until now we described only a few popular SWS approaches. However the research work in this area is quite rich and diverse. While the majority of the approaches focus on describing the semantics of the inputs and outputs or the service functionality, Cardoso, Sheth et al. [CSM02] argue that the quality of service metrics play an

important role in the context of certain service tasks such as discovery and composition. Therefore, Web services operational metrics need to be captured using a suitable model describing the QoS metrics. QoS represents the quantitative and qualitative characteristics of an e-workflow application, which are necessary to achieve a set of initial requirements. In particular, the authors introduce a model for describing execution time, cost, reliability, etc. It is important to point out that the introduced model does not contradict the approach followed in this thesis. To the contrary, it can be used as an extension to MSM, in order to complement it with QoS features.

Further SWS approaches include the *Capabilities Model* [OHE03], which is used for explicitly describing what a service or an agent can do. The capturing of Web service capabilities supports its advertising and discovery. DIANE is a framework for automating the discovery, composition, binding and invocation of services [KKRM05, KKRKS07]. The Semantic Web Services Ontology (SWSO) is part of the Semantic Web Services Language (SWSL) [BBB$^+$05], which consists of formal conceptual definitions as well as individual Web services. Its goal is to enable reasoning about the semantics underlying Web services. A complete overview of currently available SWS approaches is given in [PMZP12, CDM$^+$04], which also provides a timeline of how the individual solutions developed and a comparative analysis of the support that they provide.

Table 3.1 provides an overview of a selection of common SWS approaches, comparing the support that they offer for capturing different service semantics. This table is an adaptation of the summary available in [PMZP12], listing only the SWS approaches applied on traditional Web services and the coverage that they provide in terms of functional, non-functional, informational, behavioural and process semantics. As can be seen the functional and informational semantics are the two types that are most commonly covered. The following section focuses on describing solutions for capturing semantics as part of Web API descriptions.

3.4.1 Semantic Web API Approaches

In contrast to research in the area of SWS, which has been quite prolific, as seen by the numerous description solutions detailed in the previous section, the number of semantic frameworks targeted at capturing Web API characteristics is relatively limited. This is probably due to the fact that Web APIs have only recently achieved greater popularity and wider use, thus raising the interest of the research community. In this section we discuss the two main approaches – MicroWSMO and SA-REST, aiming to support a greater level of automation of common service tasks through employing semantics. We also consider further description languages and ontologies, including ReLL and ROSM.

MicroWSMO. MicroWSMO [KV08] is a formalism for the semantic description of Web APIs, which is based on adapting the SAWSDL approach. MicroWSMO uses microformats for adding semantic information on top of HTML service documentation, by relying on hRESTS for marking service properties and making the descriptions machine-processable. It uses three main types of link relations: 1) **modelReference**, which can be used on any service property to point to appropriate semantic concepts identified by URIs; 2) **liftingSchemaMapping** and 3) **loweringSchemaMapping**, which associate messages with appropriate transformations (also identified by URIs) between the underlying syntactic format such as XML or JSON and a semantic knowledge representation format such as RDF. Therefore, MicroWSMO, based on hRESTS, enables the semantic annotation of WebAPIs in the same way in which SAWSDL, based on WSDL, supports the annotation of Web services.

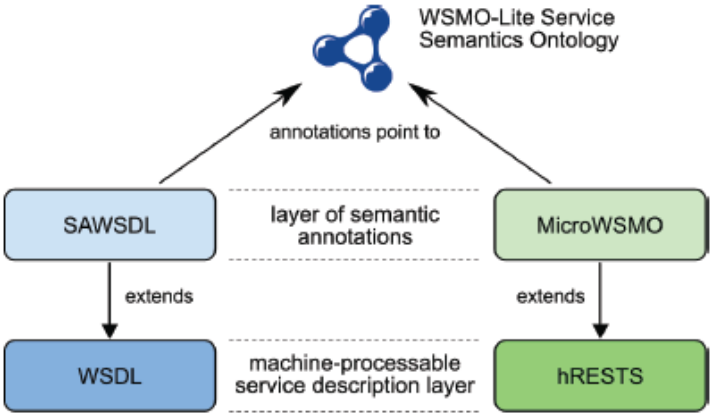


FIGURE 3.7: Unifying SAWSDL and MicroWSMO through WSMO-Lite

In addition, MicroWSMO can be complemented by the WSMO-Lite service ontology specifying the content of the semantic annotations (see Figure 3.7[16]). Since both Web APIs and WSDL-based services can have WSMO-Lite annotations, this provides a basis for integrating the two types of services. Therefore, WSMO-Lite enables unified search over both Web APIs and WSDL-based services, and tasks such as discovery, composition and mediation can be performed based on WSMO-Lite and a shared service model, such as the Minimal Service Model introduced in Chapter 7, completely independently from the underlying Web service technology (WSDL/SOAP or REST/HTTP).

SA-REST. Another formalism for the semantic description of RESTful services is SA-REST [SGL07], which also applies the grounding principles of SAWSDL but instead of using hRESTS relies on RDFa [ABMP08] for marking service properties. RDFa enables the embedding of RDF data in HTML by providing a set of attributes that can be used to insert metadata in an XML language, in general, and in an XHTML Web API documentation, in the scope of our work.

[16] Source – MicroWSMO and hRESTS [KVF09]

Similarly to hRESTS, SA-REST enables the annotation of existing HTML service documentation by defining the following service elements: service, operation, input message, parameter, output message, HTTP method. Therefore, it implicitly defines a service model based on these properties. In addition, it also includes the specification of lifting, lowering, or fault and linking these to semantic entities.

MicroWSMO, in combination with hRESTS, and SA-REST represent two very similar approaches and the main differences are not the underlying principles but rather the implementation techniques. As already mentioned, the service characteristics that can be captured by SA-REST are operation, input, parameter, output and HTTP method. Similarly to hRESTS, this accounts for the low complexity of both the description formalism and the format. However, this also affects the level of support for performing common device tasks. For instance, as is the case with hRESTS, SA-REST supports the automated recognition of the HTML documentation as a Web API description and the identification of the service properties. However, in terms of enabling invocation, there are crucial elements missing, such as the endpoint and the way of passing the input in the HTTP request. Similarly, authentication information cannot be captured with SA-REST.

ROSM. The Resource-Oriented Service Model (ROSM) [FN10] ontology is a lightweight approach to the structural description of resource-oriented Web APIs, compatible with WSMO-Lite annotations. It is a service model ontology that enables the annotation of resources belonging to a service. In turn the resources can be described as being part of collections and having addresses (URIs) serving as endpoints for access and manipulation. The organisation of resources in collections, which again belong to a service, allows capturing an arbitrary number of resources and attaching semantic annotations to them by following the SAWSDL approach.

Furthermore, resources can have certain HTTP methods associated with them, which define how it is possible to interact with a resource. The methods are realised in terms of operations that are assigned to the request. To this purpose, ROSM[17] enables the explicit modelling of requests and responses with their associated aspects (e.g., parameters, response codes, etc.). In summary, ROSM represents a simple ontology for describing resource-centred services, in terms of resources, collections of resources, addresses and HTTP methods. The specific semantics of a ROSM-based service description can be made explicit by using the WSMO-Lite ontology.

ReLL. Resource Linking Language (ReLL) [AW10] is a language describing interlinked REST resources, and thus the service that can be accessed by interacting with those resources. It is based on a service description metamodel consisting of a service that provides one or more resources that have optionally a URI pattern. The URI pattern does not represent a fixed structure for a URI but instead describes the constraints for resource unique identifiers. Each resource may have representations, which are the serialisation of the resource in some syntax. In turn, each

[17] `http://www.wsmo.org/ns/rosm/0.1/`

representation can contain links relating one resource to another target resource. The explicit description of links between resources is a unique characteristic of ReLL. The links are typed and can be retrieved from representations through selectors that can be specified, for example, through the XML Path Language (XPath)[18]. In addition, links follow the rules specified by a protocol, including the method to be used for the request, plus additional information.

In summary, MicroWSMO (and hRESTS), SA-REST, ROSM and ReLL all provide a service description model, which is used as a basis for adding semantic annotations. The models used by MicroWSMO (through hRESTS – service, operation, input, output, HTTP method and labels) and SA-REST (service, operation, input message, parameter, output message, HTTP method) are very similar and take an operation view on Web APIs. As previously discussed, they provide basic support for task automation but lack some key information, which would be required for automatically completing the invocation process. In contrast, ROSM and ReLL take a resources-based view, adopting a RESTful approach towards Web APIs. It is important to point that, even though, these approaches assume that Web APIs follow the REST principles, they still define an explicit service model, instead of relying on the architectural design decisions for interacting with the service.

In the context of providing support for the invocation task, ROSM is unfortunately not very well documented. However, it does define an HTTP method and provides support for describing how the parameters are sent in the request – via the URI, the HTTP header or the HTTP body. What is missing is the endpoint specification and a description of how the parameter values are used to construct the HTTP request. The later is not a part of the service description model but is required grounding information, which prescribes how the semantic Web API description can be actually used to construct a request that can be processed by the server. ReLL, on the other hand aims to create an interlinked network of resource representations, which can be explored. In this context, invocation is reduced to being able to follow the links and transfer from one resource, and its representations, to the next one. This is an adequate solution that works within the scope of the ReLL-based work, however, it cannot directly be applied to the multitude of existing individual Web APIs, which do not always take a resource-oriented view and for which the relevant links remain to be defined.

Further Semantic Web API Approaches. ServONT is an ontology-based hybrid approach where different kinds of matchmaking strategies are combined together to provide an adaptive, flexible and efficient service discovery environment [BAM08]. Semantically-enriched frameworks are considered key for enabling discovery and dynamic composition of services. In contrast, the ServFace project aims at creating a model-driven service engineering methodology for

[18] `http://www.w3.org/TR/xpath/`

an integrated development process for service-based applications[19]. Finally, the Simple Semantic Web Architecture and Protocol (SSWAP) is the driving technology for the iPlant Semantic Web Program[20]. It combines Web service functionality with an extensible semantic framework to satisfy the conditions for high throughput integration [GSM+09]. SSWAP originates from the Semantic MOBY project, which is a branch of the BioMOBY project [WL02]. Using SSWAP, users can create scientific workflows based on the discovery and execution of Web services and RESTful services.

Approach	**Functional**	**Non-Functional**	**Informational**	**Behavioural**	**Processes**
MicroWSMO	via WSMO-Lite	via WSMO-Lite	via WSMO-Lite	via WSMO-Lite	no
ServONT	yes	no	yes	yes	not explicitly
SA-REST	not explicitly	not explicitly	yes	no	no
ServFace	yes	no	yes	yes	yes
SSWAP	yes	no	no	yes	not explicitly
RELL	no	no	no	no	no
ROSM	via WSMO-Lite	via WSMO-Lite	via WSMO-Lite	via WSMO-Lite	no

TABLE 3.2: Semantic Approaches for Describing Web APIs

Table 3.2 lists all the semantic approaches that can be used for describing Web APIs. As can be seen, none of the approaches provide support for capturing non-functional and process semantics (the only exception is ServFace). In contrast, the majority focus on functional and informational semantics, which are also crucial in the context of providing support for the most commonly performed service tasks, such as discovery and composition. The current status of the semantic Web API description approaches demonstrates that still more advanced topics such as orchestration and choreography are not the focus of the research activities. To date, the primary concern is with identifying the type of functionality that the API provides and capturing the informational semantics of the data that it processes.

It is important to point out that by analysing the characteristics of the semantic Web API approaches it does not directly become evident that the automated completion of the invocation and authentication tasks represents a challenge, which to date remains unsolved. Invocation and authentication were never an issue in the context of traditional Web services, which rely on WSDL files and existing implementation libraries that enable the direct generation of client stubs, which can then be used to call the service. Similarly, authentication is specified and handled by the WS-Security stack. In contrast, Web APIs are all invocable directly over HTTP, however the way of transmitting the input data, the used HTTP method, the way of composing the invocation URI and the complete HTTP request, differ from API to API. This is due to the fact that Web API development proliferates autonomously and it is up to the providers to decide on the specific details of how to enable access to their resources. Therefore, the work presented in this thesis focuses precisely on tackling these issues, which remain unaddressed by current semantic Web API approaches.

[19] `http://www.servface.eu/`
[20] `http://sswap.info`

Even though, there is some initial research in the area of semantic Web API descriptions, which can be used as a foundation for further work, none of the here discussed formalisms can be directly adopted for the creation of semantic descriptions, which enable the automated completion of common service tasks. The existing formats for creating machine-processable descriptions based on available HTML documentation, in particular hRESTS, do not support the identification of all service elements commonly provided within a Web API or Web application documentation. For instance, most service descriptions include information about the necessary authentication, the response format, and examples. However, currently none of the available formats can represent this information.

3.5 Integrating Web Services and Linked Data

Linked Data (LD) is used to refer to a set of principles [BCH08] describing the way for publishing and connecting structured data on the Web [BHBL09]. Following these principles, data providers can publish and link their data, contributing to the development of a global data space – the **Web of Data**. Linked Data suggests that the same principles that enabled the Web of documents to become so popular should be applied to data as well. In this way, data coming from diverse domains, which has previously been hidden away in databases or repositories, can be made available on the Web and interlinked with existing documents.

The Web of Data provides the basis for new types of applications, which can operate on it. For example, data can be explored with the help of a browser such as Tabulator [BlCC+06], by following links to related data sources. In addition, it can be used for building Web applications [Hau09] or it can be searched and queried for a particular topic of interest by simultaneously covering a number of linked sources.

By applying the underlying principles of the Web, Linked Data uses the Web to create links between data coming from different sources. The resulting published data is machine-readable, with explicitly defined meaning, and is linked to/from other external data sets. There is a set of rules[21] defined for publishing data on the Web in a way that it becomes part of a single global data space. These rules are summarised below:

1. Use URIs to name things.
2. Use HTTP URIs to enable looking up (dereferencing) the item identified by the URI.
3. Provide useful information, when dereferencing the URI to a thing (using standards such as RDF and SPARQL [PS08]).

[21] `http://www.w3.org/DesignIssues/LinkedData.html`

4. Links to other URIs should be included, which enables the discovery of more data.

These principles should be carefully evaluated, in the context of Web APIs, since services can operate on and manipulate the data, in return providing output information, which can be published as Linked Data as well.

The main technologies behind Linked Data include HyperText Transfer Protocol (HTTP), Uniforms Resource Identifiers (URIs) and Resource Description Framework (RDF*). HTTP and URIs are essential technologies for the Web, where URIs provide the means for identifying documents and other entities and HTTP enables retrieving resources or entity descriptions based on the provided URI. These are supplemented by RDF, which provides a graph-based data model for structuring and linking data that describes things in the world.

By using these three underlying technologies and by following the Linked Data Principles, data providers can add their data to the global data space, where it can be discovered and used by applications and users alike. There are three basic steps for **publishing data sets as Linked Data** [HB11]:

1. Entities described by the data should be assigned URIs and these URIs, when retrieved over HTTP, should result in the corresponding RDF representations.
2. RDF links to other data sources on the Web should be defined.
3. The published data should be supplemented by metadata.

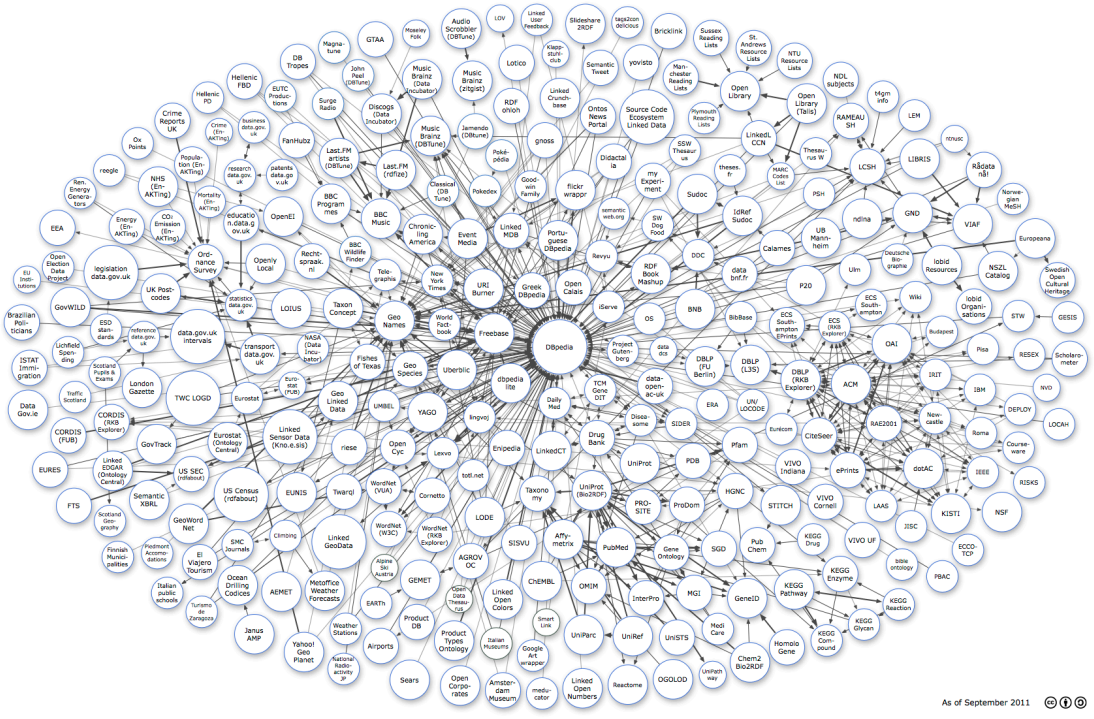
More detailed guidelines about how to publish Linked Data on the Web are available in [BCH08]. By following these steps for adding datasets and adopting the Linked Data Principles, the W3C's **Linking Open Data**[22] (LOD) project was developed.

Figure 3.8[23] illustrates the datasets participating in the LOD cloud. This initiative started with only a few datasets and more and more datasets were linked, leading to the bootstrapping of the Linked Open Data cloud. In general, anyone can publish a dataset by following the Linked Data principles. Currently, the cloud contains diverse information, including data about people, scientific publications, books, geographical locations, music, films, and many more.

The published data can be used as a basis for developing a wide variety of new applications, which can dynamically analyse and interpret the data, browse from one data source to another by following the links, or collect and query data related to a particular topic of interest. There are two main types of Linked Data applications. The first ones are data interaction applications, such

[22] `http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`

[23] Source – Linked Data - Connect Distributed Data across the Web, `http://linkeddata.org/`, last updated: 2011-09-19


DBpedia
Freebase
Geo Names
Linked MDB
Event Media
Chronicling America
BBC Music
BBC Programmes
OpenEI
Openly Local
Ordnance Survey
legislation data.gov.uk
UK Postcodes
data.gov.uk intervals
statistics data.gov.uk
LOIUS
Taxon Concept
Geo Species
Fishes of Texas
Uberblic
Geo Linked Data
LIMBEL
YAGO
lingvoj
Open Cyc
Lexvo
riese
Linked Sensor Data (Kno.e.sis)
Eurostat (Ontology Central)
GovTrack
US SEC (rdfabout)
US Census (rdfabout)
EUNIS
Twarql
WordNet (VUA)
WordNet (W3C)
WordNet (RKB Explorer)
Cornetto
LODE
SISVU
Affymetrix
PubMed
Gene Ontology
SGD
Pub Chem
KEGG Drug
KEGG Enzyme
KEGG Pathway
KEGG Glycan
KEGG Reaction
KEGG Compound
Homolo Gene
GeneID
MGI
InterPro
UniSTS
UniRef
UniParc
Reactome
OGOLOD
Linked Open Numbers
Amsterdam Museum
Open Corporates
Sears
Airports
Weather Stations
Product DB
Product Types Ontology
Linked Open Colors
GEMET
AGROVOC
ChEMBL
OMIM
Linked GeoData
Metoffice Weather Forecasts
AEMET
Ocean Drilling Codices
Janus AMP
Yahoo! Geo Planet
Linked Edgar (Ontology Central)
Semantic XBRL
GeoWord Net
Eurostat
TWC LOGD
London Gazette
data.gov.uk intervals
transport data.gov.uk
Lichfield Spending
ISTAT Immigration
Data Gov.ie
CORDIS (FUB)
CORDIS (RKB Explorer)
EURES
FTS
Scotland Geography
Finnish Municipalities
Italian public schools
Tourism de Zaragoza
El Viajero Tourism
Piedmont Accommodations
SMC Journals
Climbing
Aspire Ski Austria
Open Data Thesaurus
Smart Link
Italian Museums
Google Art wrapper
medu-cator
Product Types Ontology
Janus AMP
Yahoo! Geo Planet
National Radioactivity
Weather Stations
Airports
Sears
Open Corporates
Amsterdam Museum
Linked Open Numbers
Reactome
OGOLOD
Unlock way
Chem2 Bio2RDF
Homolo Gene
KEGG Compound
KEGG Glycan
KEGG Reaction
KEGG Enzyme
KEGG Pathway
KEGG Drug
VIVO UF
Pub Chem

include the BBC Sports[39], ResearchSpace[40] and Open Pharmacology Space[41]. However, overall the applications hardly go beyond aggregating, filtering and browsing data gathered from different sources, leaving the potential of this massive data space mostly unexploited.

More importantly, in the context of Web APIs and services on the Web in general, the proliferation of Linked Data clearly demonstrates how lightweight semantics can bring significant benefits, which, in turn, justify the efforts that need to be devoted to making the annotations and deploying the necessary machinery. Furthermore, this initiative is contributing to generating a body of knowledge, in the form of lightweight ontologies and data expressed in their terms, that can help to significantly reduce the effort for creating semantic annotations for services. Therefore, Linked Data can be used as background information, for automatically suggesting service annotations, but it also already provides a set of ontologies, which can directly be used for semantically describing services. The resulting semantic descriptions of Web APIs can be published by following the Linked Data Principles and be directly integrated in the Linked Data cloud [PD10].

Some of the main approaches aiming to contribute to the integration of services on the Web and Linked Data are described in more detail below.

The first step in this direction was made by implementing Linked Data interfaces for services, e.g., in the form of the book mashup [BCG07], which provides RDF about books based on Amazon's API, or twitter2foaf[42], which encodes a Twitter follower network of a given user based on Twitter's API. These are useful examples for the integration of information services and Linked Data. However, they are based on manual implementation work and do not present an overall solution. Still these initial efforts are valuable contributions both from the point of view of the offered dataset as well as demonstrating the benefits of integrating Web APIs and Linked Data.

Linked Data Services. Linked Data Services (LIDS) [SH10], are part of the new trend on Linked Services [PD10], which aim to bridge the gap between Linked Data and services. LIDS focuses on integrating existing data services (e.g. services that provide data) exposed through Web APIs, with Linked Data principles [BHBL09] by having LIDS consume and produce RDF triples. In particular, Web APIs are described with a lightweight service description model where service inputs and outputs are specified using SPARQL graph patterns. The description model is defined by a simple LIDS vocabulary[43], which defines a class for LIDS and a property

[39] BBC Sports – Dynamic Semantic Publishing, `http://www.bbc.co.uk/blogs/bbcinternet/2012/04sports_dynamic_semantic.html`

[40] ResearchSpace – a digital Wunderkammer, `https://sites.google.com/a/researchspace.org/researchspace/`

[41] Open Pharmacy Space Platform, `http://www.openphacts.org/open-phacts-discovery-platform`

[42] `http://www.alexa.com/siteinfo/twitter2foaf.appspot.com`

[43] `http://openlids.org/vocab`

relating a LIDS to a SPARQL query by using the CONSTRUCT operator. A Linked Data Service invocation is equivalent to running the so defined SPARQL query, where the input is defined in the WHERE selector, in the form of a simple graph-pattern, the endpoint is within the FROM operator and is the base URI of the service, and the resulting *io-relation* relates one of the input variables to the output variables. The approach does not explicitly define a service model, but rather encapsulates a Web API in a wrapper that assumes the existence of input, endpoint and output. Authentication needs to be handled separately, since LIDS does not provide support for describing an authentication mechanisms or submitting credentials.

In summary, LIDS are parameterised and formally described Web resources, which return RDF when dereferenced via HTTP. The difficulty here lies in creating the actual Web API wrappers, which currently have to be written manually by interpreting the existing HTML documentation. More importantly, the approach requires an architectural shift where descriptions are seen as SPARQL queries, which can be run against a global data space. The benefits of investing effort into creating annotations and providing the required processing machinery still remains to be seen.

LOS. Another approach targeted at supporting the integration of Linked Data and Web APIs is Linked Open Services (LOS) [KNM10]. LOS aim to provide support for combining LOD endpoints and Web APIs in general. In this context, services are described based on RDF and SPARQL, and show how each service is a consumer and producer of RDF, by defining inputs and outputs as SPARQL graph patterns. Existing work on linking service messaging with semantic representations. e.g. lifting and lowering [KV08], is reused in order to describe how the RDF input is mapped to the particular values and, in turn, how the produced output can be transformed into RDF. The followed approach is similar to LIDS, where the original Web API interface is wrapped in a new description. In the case of LOS, the wrapper includes the inputs and outputs in terms of the expected/produced RDF (instead of, for example, Strings and XML/JSON). The used service model is based on the one provided by hRESTS, with annotations of the inputs and outputs. LOS does not include any specific work on implementation and authentication and the required activities have to be implemented as part of the wrapper.

Current work in this filed also provides initial approaches towards supporting the creation of Linked Services. Taheriyan et al. [TKSA12] (based on work on Karma [ADG+09]) presents a solution that allows domain experts to create semantic models of services with the help of an interactive web-based interface. In particular, the work focuses on semi-automatically building semantic models that can be used for the annotation of Web APIs, including the lowering and lifting specifications. Based on samples of Web API requests and a set of vocabularies, the system invokes the service and creates a model that captures the semantics of the inputs, outputs and their relationships. The results are visualised in a graphical interface that can also be used to make corrections. In addition, the system also generates the lowering and lifting specifications.

The service model in this approach is based on using SWRL[44] to define the input and output model, and the Karma knowledge model[45]. In addition to creating the semantic linked service descriptions, the system provides support for storing them, searching with the help of SPARQL queries, and invoking them. The main limitation of this work is that the solution allows using only the HTTP GET method and is restricted to assuming that all inputs for service invocation are embedded in the invocation URL. There is no clear description of how authentication is handled it this context.

Saquicela et al. present a semi-automatic approach for annotating Web APIs in the geospatial domain [SBC12, SBC11]. In particular the work focuses on creating syntactic descriptions of Web APIs and semantically enriching their parameters, through the partial automation of the process. Therefore, we discuss this approach in more detail also in Chapter 5 on Annotation Approaches and Tools. However, the authors also introduce a service description model based on – method (not denoting the HTTP method but rather the operation), input and output, as well as invocation, input value and output value. As a result the description contains both the service properties, as well as the values that would be required for the invocation. Semantic information from DBpedia[46] and GeoNames[47] is used to enhance Web APIs with metadata through defining *Parameters* that represent the semantic annotations of inputs and outputs. The presented work is not especially focused on supporting invocation and authentication, therefore, these are not discussed in much detail.

A final approach that needs to be mentioned is offered by Verborgh et al. [VSD+12] who introduce *RESTdesc*, as a means for capturing the functionality of hypermedia links in order to integrate Web APIs, REST infrastructure, and the Linked Data. The idea is to enable intelligent agents to get additional resources at runtime from the functional description of the invoked API. RESTdesc uses N3 notation to express the service description. Similarly to LIDS, LOS, and Taheriyan et al., it can model the relationships between the input and output attributes.

In summary, there are currently two main groups of approaches aiming to enable the integration of Web APIs with the Linked Data cloud. The first one focuses on annotating the service attributes using concepts of known ontologies and publishes the service descriptions into the cloud [PD10, ADG+09]. The second one is based on wrapping the APIs in order to enable them to communicate at the semantic level so that they can consume and produce Linked Data [SH10, KNM10]. These approaches represent an important milestone towards enabling the integration of Web APIs and the Web of Data, however, they do not directly offer a solution to task automation problems, especially focusing on invocation and authentication.

[44] Semantic Web Rule Language, `http://www.w3.org/Submission/SWRL/`
[45] `http://isi.edu/integration/karma/ontologies/model/current`
[46] The DBpedia Ontology, `http://dbpedia.org/Ontology`
[47] GeoNames Ontology, `http://www.geonames.org/ontology/documentation.html`

3.6 Summary

This chapter provides an overview of existing research work related to creating semantic Web API descriptions. We draw a parallel between traditional Web services and Web APIs by exploring the existing solutions that support individual steps or the complete lifecycle of services, with or without the help of semantics. In particular, we summarise current approaches towards syntactically describing Web services and Web APIs. These approaches lay the foundation for the employment of semantics as a means for providing a higher level of abstraction that enables service processing based directly on individual service characteristics, instead of on technical details. We discuss common SWS frameworks and some semantic Web API solutions. We provide a comparative overview of the types of semantics that each approach covers and identify gaps in the corresponding solution support.

In summary, current research in the area of SWS and semantic Web API descriptions provides a foundation, which can be extended by approaches and tools particularly targeted at handling the challenges faced by Web APIs, which have up to date remained unaddressed. Developments in the area of Web API invocation and description are marked by two main trends. The first one is simplicity – visible through the autonomous development based on a simple technology stack (URIs and HTTP) and a simple, widely accessible description form (HTML webpages). This trend is recognisable also in the existing semantic Web API approaches, which are very lightweight and not as complex as solutions in the context of SWS, such as WSMO and OWL-S. The second trend is the integration of Web APIs with Linked Data – demonstrated by the growing number of mashups and applications[48] created on top of Web APIs and the increasing efforts towards bridging the gap between Web APIs and Linked Data.

Since the approach that we follow is based on developing a semantic Web API description model, which enables the automation of the invocation and authentication tasks, we give a summary of the existing service description models and the level of support that they provide. Table 3.3 includes an overview of the approaches discussed in this chapter, giving details on whether they include a service description model, if it is operation- or resource-based, if there is any work related to supporting the invocation, how is the grounding realised and which service properties are enhanced with semantic information. It is important to point out that some of the discussed solutions actually offer only a model (for example, ROSM), while others propose approaches (LIDS) and complete systems (Taheriyan et al.). Still we compare these based on the models that they use to specify what a service is and the resulting task support.

As can be seen, there are different ways of approaching the definition of a service description model. For some of the offered solutions the model is the essential contribution, while for others

[48] Currently there are 6812 mashups and Web applications and their number is constantly growing, source – programmableWeb, `http://www.programmableweb.com/mashups`, viewed 10.10.2012

Approach	Explicit Service Model	Operation-based	Resource-based
MicroWSMO/hRESTS	MSM	yes	no
SA-REST	SA-REST Model	yes	no
ROSM	ROSM	no	yes
ReLL	ReLL metamodel	no	yes
LIS	implicit	yes	no
LOS	MSM	yes	no
Taheriyan et al.	SWRL&Karma KM	yes	no
Saquicela et al.	yes	yes	no

Approach	Invocation Support	Grounding	Semantic Annotations
MicroWSMO/hRESTS	insufficient details	lifting/lowering	via MicroWSMO
SA-REST	insufficient details	lifting/lowering	yes
ROSM	insufficient details	unclear	via SAWSDL/MicroWSMO
ReLL	implicit via defined links	NA	no
LIS	via SPARQL and wrapper	as part of the wrapper	on input and output
LOS	via wrapper	lifting/lowering	on input and output
Taheriyan et al.	limited support	lifting/lowering	on input and output
Saquicela et al.	test invocation	explicit mapping	on input and output

TABLE 3.3: Service Description Models for Web APIs

it is a means to supporting a certain task. However, for all of the here discussed work it is essential to specify what the particular Web API characteristics are and what makes out a service, since theses provide the foundation for the developed approaches. This also indicates that relying on the REST architectural principles is insufficient and formally determining the relevant service properties gives the necessary basis for developing solutions related to processing and interacting with Web APIs. In summary, as shown by the overview of existing approaches, all solutions related to automating the use of Web APIs to some extent rely on a service description. Furthermore, currently most of the solutions take an operation-based view on APIs, which demonstrates that RESTful services are not central both in the context of designing APIs and in providing approaches and tools that support their use.

In the context of invocation, most description models do not include some of the information that is actually required, such as SA-REST and MicroWSMO, while others hardcode the solution as part of the implementation of the wrappers. Currently the most extensive invocation solution is provided by Taheriyan et al. and it is limited to Web APIs that can directly be invoked via HTTP GET and a parameterised URI, which basically reduced the work of an invocation engine to creating and processing the GET HTTP request and response. The grounding support is frequently realised through the definition of lifting and lowering transformations, and the semantic annotation of service properties is often limited to inputs and outputs.

Overall, neither the service description models, nor the existing invocation solutions provide support with wide coverage, which can be used to deal with the majority of the currently available APIs. The description models miss key service characteristics or are suited only for a subset of the Web APIs. Similarly, the invocation support is limited to custom implementations or to very basic cases. Furthermore, none of the discussed solutions consider authentication support.

The related work clearly demonstrates that in the context of Web APIs, there still remain quite a few challenges that are unaddressed by existing approaches. This is especially true for providing support for completing the invocation and authentication tasks. The chapter is concluded by a discussion of current trends in the area of Web API development.

In the following chapter we explore work related to Web API invocation and authentication, and use it as a foundation for deriving further objectives for developing an approach towards creating semantic Web API descriptions.

Chapter 4

Invocation and Authentication Approaches

In the context of developing service-oriented applications, the discovery of services, ranking and selection of the results, and the creation of compositions are fundamental but also time consuming tasks. However, none of the tasks influence the successful usage of Web APIs as much as providing proper invocation support. Even if a suitable Web API is known or found and the data required to make a service request is available, the API cannot be actually used without support for invoking it. While for traditional Web Services (WS) based on the WSDL description, invocation can be handled directly without extensive manual effort, for most APIs the server requests and responses have to be realised through individual implementation solutions. The same holds for retrieving data based on a composition of Web APIs. Some providers address this difficulty by offering individual clients for using a particular Web API, however, this approach has rather limited coverage and the majority of the Web APIs require that developers interpret the existing documentation and implement customised invocation solutions. Therefore, in this chapter we focus on exploring existing approaches that support Web service and Web API invocation. Furthermore, many providers restrict the access to their services by requiring some form of authentication. Actually, more than 80% of the APIs require authentication (see Chapter 6), a fact that is commonly ignored by existing Web API software infrastructures. Therefore, we also provide an overview of commonly used authentication approaches in the context of Web API invocation.

In particular, this chapter contains two main sections – one on invocation approaches and solutions, and one on authentication. As in the previous chapter we explore work that has already been done in the context of traditional Web services and draw a parallel to the solutions devised especially for handling Web APIs. In particular we analyse invocation approaches in terms of:

- The type of service that is supported;

- The level of support;
- In the case of semantic solutions, the ways of handling lifting and lowering.

We conclude with a brief discussion of the current state of invocation and authentication solutions and derive a set of objectives that are to be met by the approach followed throughout this thesis.

4.1 Web API Invocation

Automated invocation in the context of "traditional" Web services has not been prominently addressed by semantic Web service researchers because of the solutions and libraries that provide direct support based on WSDL. In particular, invocation done directly on the level of WSDL and SOAP is well supported by a variety of frameworks and implementations (see following section). Invocation based on semantically annotated WSDL files is handled with the help of lifting and lowering transformations. In this section we give an overview of the main invocation tools and describe existing efforts towards supporting the dynamic invocation of Web services, which presents the challenge of identifying the services to be invoked at runtime. We continue by identifying the main reasons that hinder us from adopting invocation solutions that were devised in the context of Web services to also enable Web API invocation and, therefore, focus in particular on research targeted specifically at enabling Web API invocation. Finally, we conclude this section by analysing the invocation support provided by existing Web API description models.

4.1.1 WSDL-based Web Service Invocation

Web services based on WSDL and SOAP have for a long time dominated the world of services on the Web, even though, they are currently being overshadowed by the proliferation of Web applications and APIs. The invocation of Web services relies on the information provided in the WSDL documentation that allows for the decoupled definition of the abstract functional characteristics of a Web service and their specific binding to a particular implementation. The part describing the specific implementation defines how the service can be accessed using a particular communication protocol, commonly SOAP, including the service endpoints and their protocol bindings (for more details on WSDL see Section 3.1).

Invocation based on WSDL is relatively well supported, even though, the implementation support provided for WSDL 2.0 [W3C07a] is somewhat more limited and offered by fewer libraries than the one for WSDL 1.0 [W3C01]. In this section we focus mostly on discussing existing

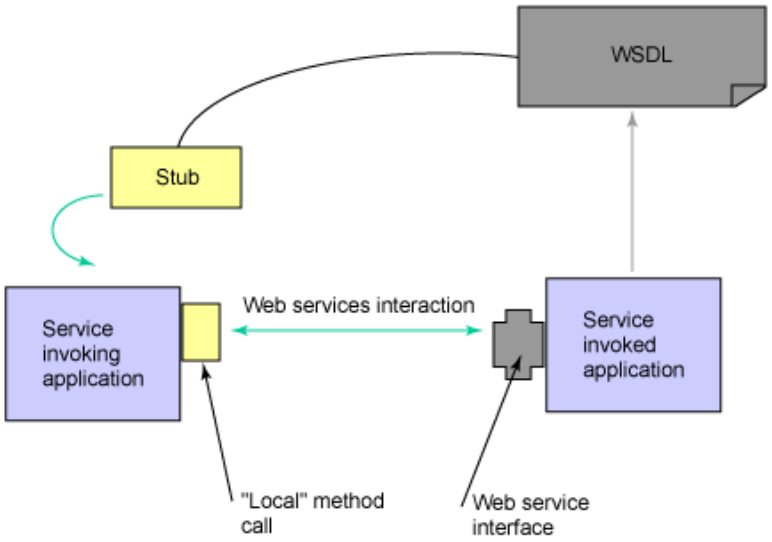


FIGURE 4.1: Automated WSDL-based Invocation

solutions based on Java, since it provides the richest and most commonly used implementations, and because this is the programming language that we used for developing our own applications. Library implementations based on WSDL such as Apache Axis[1] or Apache Web Services Invocation Framework (WSIF) [DMS+01][2] take the Interface Description Language (IDL) file and generate the necessary wiring such as stubs and implementation skeletons or templates, for implementing a Web service client within a specific programming language. Further implementation solutions include GLUE[3], SOAP::Lite for Perl[4] and simple tools such as Generic SOAP Client[5] that directly support the SOAP message generation. Figure 4.1[6] shows the general approach towards automated WSDL-based invocation [Cer02], where the complexity of marshalling parameters and executing remote procedure calls (RPC) is hidden from the programmer and the generated code overcomes the necessity to have any knowledge or to manipulate the raw SOAP or XML messages. In particular, the stub code on the client side is automatically generated based on consuming an interface description of the invoked function, expressed in WSDL. The stub wraps the function from the server side and offers it as a local function to the client. Therefore, based on the operations, inputs, outputs, their types, service ports/endpoints and specific bindings described in a WSDL file and benefiting from the existing tool implementation, Web service invocation based on WSDL can be performed automatically.

Dynamic WS invocation. However, the task of invoking Web services becomes more challenging when the necessity arises to be able to dynamically change service providers instead of

[1] `http://axis.apache.org/axis2/java/core/`
[2] `http://ws.apache.org/wsif/`
[3] `http://www.themindelectric.com`
[4] `http://www.soaplite.com`
[5] `SOAPClient.com`
[6] Source – SOA adventures, Part 1: Ease Web services invocation with dynamic decoupling [Dav05]

hardwiring them via the generation of stubs at design time. The majority of current Web service client frameworks rely strictly on pre-generated service invocation code on the client-side that is not exchangeable at run time. In that sense, it can be argued that if service providers are hardwired into the service consumers application code, producers and consumers cannot by any means be considered loosely coupled resulting in the need to provide support for dynamic service invocation [LRD09].

There are a number of approaches and implementations targeted at supporting at least to some extent the dynamic invocation of Web services. Apache WSIF provided the first solution incorporating the idea of dynamic service invocation [DMS+01]. It is a step in the right direction, even though the possibilities for exchanging service providers at run time is somewhat limited. In particular it requires that the exact signature of the WSDL operation to invoke is known in advance and has some obstacles when it comes to using complex XML Schema types as service parameters or return values because they require additional coding and need to be mapped to an existing Java object beforehand.

A more extensive solution is provided by the Apache Axis 2 [The12a] framework that offers more flexibility and more options for realising the invocation than WSIF. Axis 2 supports dynamic invocation through a set of APIs but requires the client to create the complete SOAP body and to craft the input for the requests itself. It supports client-side asynchrony and despite the fact that it still depends on the usage of stubs, it enables the swapping of services at run time, if the client provides the entire payload for the invocation. In addition to supporting SOAP, it also has a well developed REST stack. Similarly to Axis 2, Codehaus XFire [Cod08] and its successor – Apache CXF [The12b], rely on static and pre-generated components to access Web services, therefore, catering for little support for dynamic invocation scenarios.

JAX-WS (Java API for XML-based Web Services) [KG07], the followup of JAX-RPC [JSR03] is implemented as part of the Apache CXF project, focusing mainly on WSDL-to-operation mappings. It also provides HTTP binding support, in addition to SOAP, so that the communication can be realised directly over HTTP. Therefore, one can directly send XML over HTTP. In this sense, for a restricted number of Web APIs with a WSDL description, which rely on XML for representing the inputs and outputs, this framework would constitute an implementation solution.

Finally, in terms of enabling dynamic service invocation, a different approach has been proposed by Nagano et al. [NHOH04], who rely on using static stubs but provide more flexibility by binding them to more generic interfaces instead of service-specific ones. Therefore, one stub can be used to invoke any service with a similar interface. BPEL [MVD12] enables the dynamic swapping of a WSDL file by the using the *Partner Link* element. If an exchange is made, the new WSDL files is automatically wrapped and integrated within the current process. The DAIOS [LRD09] framework represents a main contribution by providing a client-side service

framework that features a fully expressive and easy to use dynamic interface, works entirely message-oriented, has full support for non-blocking communication and supports SOAP-based services and Web APIs. In the context of Web APIs, the user can specify an example request instead of the WSDL file. This example is used to generate a parameterised URI, which is filled with the corresponding values in order to perform the invocation.

Up to now we have discussed invocation approaches, which are based on WSDL descriptions. As can be seen, the support in the context of WSDL in combination with SOAP is quite extensive. This includes a number of systems that also support dynamic invocation. In contrast, there are no invocation solutions that have been solely designed to support Web APIs. Instead the systems that provide some level of support, offer this as an additional functionality. Within JAX-WS, Web APIs can be invoked based on XML messaging and HTTP communication, given a WSDL description. DAIOS represent the most advanced solution in the context of Web APIs and enables the invocation based on parameterised URIs. Dynamic invocation, specifically in the context of Web APIs, is still mostly out of the research scope. This is mainly because of the difficulties faced when trying to provide some level of support for individual API invocation, without even considering the dynamic elements. In summary, existing systems that support Web service invocation based on WSDL are unsuited for being directly adopted for Web APIs. This is mainly because the solutions do not explicitly focus on Web APIs and as a result the support is limited to a particular subset, thus not providing wide coverage or a general solution.

4.1.2 Semantic Web Service Invocation

As already introduced in Section 3.4 the main objective of research in the area of Semantic Web Service (SWS) technologies is to enable the automation of common service tasks such as discovery, composition and invocation by providing ontology-based semantic descriptions for each service, comprising a formal description of the service's functionality, non functional aspects, and external behaviour. Specifically in the context of SWS invocation WSMX [FKZ08] and IRS-III [DCG+08] represent two main environments for invoking services described through the WSMO paradigm. In the context of OWL-S based approaches, the OWL-S virtual machine [PASS03] and the OWL-S API [SP04] enable the invocation of OWL-S based services.

The Web Service Execution Environment (WSMX) [FKZ08] is the result of the substantial effort of the SWS research community to develop engines and frameworks able to interpret and handle WSMO descriptions. It represents one of the reference WSMO-based execution engine implementations and is also one of the implementations of Semantic Execution Environments (SEE) being standardised within OASIS [NPDZ08]. WSMX is a SEE that aims to support invocation, as well as discovery, composition, mediation and selection, based on a set of user requirements.

Each layer supports tasks with an increasing level of complexity and addressing different stakeholders. The invocation support is implemented as part of the middle layer, which provides brokering facilities necessary for manipulating semantic Web service descriptions. This layer comprises components for discovery, selection, data and process mediation, choreography, orchestration, grounding, and transport handling. The execution environment was not specifically developed to support invocation but is rather designed around a set of different execution scenarios such as goals achievement and service selection. As a result somewhat heterogeneous components with different levels of maturity are bound via common interfaces and a shared overall architecture, aiming to support a number of predefined Web service use cases.

Another service execution environment is the Internet Reasoning Service (IRS-III) [DCG+08] that supports service invocation, among other tasks, by serving as a broker-based platform that mediates between clients and service providers, thus enabling their integration. The platform incorporates the WSMO conceptual model and provides additionally a set of tools to support the SWS developer at design time in creating, editing and managing a library of semantic descriptions as well as publishing and invoking Semantic Web Services [DCG+08]. Since the IRS-III is aligned with WSMO, it is based on contemplating *Goals*, *Web Services*, *Mediators*, as well as *Choreography* and *Orchestration* of services. In IRS-III the Web service invocation is capability driven, based on the underlying goal-centric invocation mechanism. Therefore, a client application can simply ask for a goal to be solved and IRS-III selects an appropriate service, invoking the associated Web service in turn. The grounding declares the operations involved in the invocation (communication primitives) and the associated mappings to the implementation level. More specifically, each operation input and output is associated with a lifting or lowering function. The grounding also relates to information about the corresponding publishing platform. The platform is not restricted to handling only WSMO-based services but has an OWL-S import mechanism [HDM+04]. In addition, IRS-III is interoperable with further WSMO implementations, including WSMO studio [DSK+07] and WSMX, facilitated through a common API.

The OWL-S Virtual Machine (OWL-S VM) [PASS03] provides a general purpose Web service client for the invocation of a Web service based on the process model provided by OWL-S. The main functionality offered by the OWL-S VM is to control the interaction between Web services based on the process description, captured using OWL-S. In particular, the OWL-S VM executes the process model of a given service by going through the process model while respecting the OWL-S operational semantics and invoking individual services represented by atomic processes. During the execution, the OWL-S VM processes inputs given by the requester and outputs returned by the provider services, handles the control and data flow of the composite process model, and uses the grounding to invoke WSDL based web services when needed. The grounding can be created with the help of WSDL2OWL-S [PASS03] that provides transition between WSDL and OWL-S, where the results of this transformation are a complete specification

of the grounding and partial specification of the process model and profile of OWL-S. In summary, the OWL-S VM is a generic execution engine, which can be used to develop applications that need to interact with OWL-S Web services.

OWL-S API [SP04] is a Java API that supports the majority of the OWL-S specifications. It provides an API for programmable access to create, read, write, and execute OWL-S atomic as well as composite services. The implemented library provides a basic process execution engine that can execute atomic processes and composite processes. It has been used as the basis for developing an execution framework for Semantic Web Services, based on service execution agents (SEA) [LB07]. SEA use context information to adapt the Semantic Web Services execution process to a specific situation, thus improving its effectiveness and providing a faster and better service to its clients.

One further solution addresses the semantic Web service invocation problem by employing a process execution engine. MWSCF – METEOR-S Web Service Composition Framework [SMSV05], is a framework for composing semantic Web processes. The composition is based on Semantic Process Templates (SPTs), which enable the semantic definition of each activity involved in a process. With an SPT, an executable process can be generated, where each activity is bound to a concrete Web service implementation, which conforms to the semantics of the activity. The mapping between a process and supporting Web services is based on the semantic information of the template and the semantic annotations in SAWSDL, on the Web service side. The SPTs, with the mapped Web services are used to generate an executable process specified in BPEL4WS (Business Process Execution Language for Web Services) [CGK+03], which can in turn be executed in any BPEL execution engine. This approach shows how SWS invocation can be solved as part of process composition solutions.

Similarly, Cardoso et al. [CS03] address the challenge of the composition of workflow processes based on Web Services. This work is also a part of the METEOR-S project, which builds on earlier experiences in developing a METEOR workflow management system with the emerging Web Services and Semantic Web technologies. The developed approach targets to enable the efficient discovery of Web services based on functional and operational requirements, and to facilitate their integration as part of executable processes. However, in contrast to previous solutions, it takes into account QoS aspects such as execution time, cost, and reliability [CSM02].

In summary, in the context of supporting the invocation of SWS there are some quite comprehensive systems that not only take a semantic Web service description, perform the lifting and lowering to a specific grounding, for processing the actual service request and responses, but also support more complex tasks such as choreography and orchestration. There are solutions that support OWL-S, WSMO and SAWSDL and all the frameworks take care of the data transformations (i.e. lifting and lowering) based on the provided semantic descriptions. Overall, the SWS invocation is covered by comprehensive solutions that have a high level of support.

However, their direct applicability in the context of Web APIs is doubtful. First of all the required input is semantically annotated WSDL files, which as we have already seen, are not really suitable for describing Web APIs. In addition, WSDL, to start with, is not widely adopted for describing Web APIs. In fact, none of the APIs that we analysed (see chapter 6) had a WSDL of WADL file. Finally, the discussed solutions are not really meant for a Web environment, in the context of supporting the development of client applications. In the following section, we present work related directly to Web API invocation.

4.1.3 Web API Invocation

This section provides an overview of Web API invocation approaches. We differentiate between approaches for Web APIs based on the REST principles, semantic approaches, and approaches based on platforms and mashup frameworks. Furthermore, in section 4.1.1 we already covered two approaches that partially support HTTP and URI-based Web API invocation, as part of WSDL-based frameworks.

Currently, when a developer wants to use a particular Web API he/she has to find the documentation, read through it and interpret it, create a custom client application, which has to be changed every time that the interface is changed, and which is rarely reusable or compatible with further invocation implementations. This situation is due to the fact that the majority of the APIs on the Web are described in natural language, without conforming to any particular guidelines about the format or the level of detail. As a result all descriptions are structured differently, contain different information, and sometimes even miss crucial elements such as the used HTTP method (see Chapter 6). Therefore, the development of a common shared approach that supports invocation and provides wide coverage, given the current heterogeneity, presents a challenge. As we have seen in this chapter, this situation is aggravated by the fact that most approaches that provide a Web API description model actually do not cover the information necessary to support invocation.

In this section we differentiate between approaches that provide invocation support on the level of implementation and syntactic processing, and approaches that enable automation through the use of metadata. On the syntactic level, the creation and processing of HTTP requests and responses, defining how the input is transmitted and how the output is interpreted are required. The semantic level requires the lifting and lowering of the data and the specifying of the communication protocol with the server. However, it enables processing based on the meanings of the inputs and the outputs, abstracting from the particular format, providing a higher level of automation in comparison to the support of systems based on enabling the construction of the HTTP messages.

RESTful Invocation. The automated invocation of Web APIs would not present such a challenge if all APIs would strictly follow the REST architectural design principles [Fie00]. RESTful APIs are based on resources, which are identified by global IDs – typically using URIs (see Section 3.2). The manipulation of resources or collections of resources is usually done over the HTTP methods (GET/ POST/ PUT/ DELETE) and the format of the output is determined via content negotiation over the supported media types. Unfortunately, as our survey of Web APIs and their characteristics [MPD10a] points out, currently about one third of the APIs are resource-centred and the majority are not truly RESTful (see Chapter 6). Therefore, RESTful invocation and Web API invocation, in general, need to be supported by more extensive tooling. REST specifies how the API is realised and how the communication with it should be conducted. However, when it comes to finding suitable services, creating compositions and invoking them, this specification is not sufficient in order to enable a fully-automated interaction.

The Web Application Description Language (WADL) (see Section 3.3) [Had06] is especially designed for describing RESTful services, capturing interface details based on the exposed resources, the access endpoints, the available HTTP methods, and the corresponding request parameters and response representations. WADL use is backed up by development support such as the one provided by JAX-RS[7], which is a Java API that provides support for creating Web services according to the REST architectural pattern. JAX-RS uses annotations, to enable the development and deployment of Web service clients and endpoints, which can be realised directly by using its Jersey[8] implementation. Another example is soapUI[9], which supports invocation as part of functional testing via a graphical interface. However, despite the fact that WADL could be the solution for eliminating the current heterogeneity of documentation content and structure, it is unfortunately rarely used and providers prefer to describe Web APIs directly in HTML. In fact, a Google search for "filetype:WADL" delivers only about 200 results[10], not all of which represent actual services. Furthermore, it is suitable for describing only about one third of the Web APIs, namely, the ones that are defined in terms of resources and not operations [MPD10a].

"Pipe"-based and Platform-based Solutions. The most widespread Web API invocation solutions are based on the "pipes"-approach, where a user interface allows for composing a set of services, which are pre-hard wired into the mashup framework. Example systems include Yahoo Pipes, Google Mashup Editor currently available as part of the Google App Engine, DERI Pipes, and more extensive solutions like IBM Mashup Center[11]. DERI Pipes provides a graphical environment for transforming and combining data on the Web. It supports RDF, XML and JSON, where the data aggregation is done with the help of scripting languages such as SPARQL and

[7] `http://cxf.apache.org/docs/jax-rs.html`

[8] `http://jersey.java.net/`

[9] `http://www.soapui.org`

[10] Search done on 30.08.2013, with the first page of matches being Facebook pages of people whose family name is Wadl.

[11] `http://pipes.yahoo.com/pipes/`, `http://code.google.com/appengine/`, `http://pipes.deri.org/`, `http://www-01.ibm.com/software/info/mashup-center/`

XQUERY. The result of the "Pipeline" is an output stream of data (e.g. XML, RDF, JSON) that can be used by further applications. When invoked directly in a browser, the user is presented with a standard GUI for entering the parameter values and browsing the results. The drawbacks of using such a system are the lack of flexibility, in terms of the fixed client application, its user interface and the available functionality, which is predefined by the framework implementation. Furthermore, additional Web APIs can not always be integrated into the framework and if that is possible, there is some work involved in adding the particular service in the form of coding a wrapper and implementing the framework interfaces for realising the communication.

Web API invocation is handled as part of service composition in [Pau09], where BPEL is extended in order to provide RESTful service support. A dynamic invocation approach is presented in [CLL+10], where the research is based on the introduction of the new HTTP binding in WSDL 2.0 as a promising approach for wrapping a RESTful service and then describing its interface using the WSDL language [Chi07]. However, from a practical point of view, a wide adoption of such an approach is yet to be seen [Pau09]. The newest trend in the area of API invocation is marked by the development of platforms that support only a predefined set of APIs, such as the solution offered by the *Try it* function of the Google APIs Discovery[12] or by the test invocation of Mashery[13]. These implementations are a step in the right direction but the coverage that they provide is very limited in term of the number of Web APIs that they support and the particular type of service description that can be added.

Semantic-based Approaches. To the best of our knowledge, to date there is no comprehensive solution that enables automated invocation of the majority of the APIs on the Web. Currently invocation is realised through the development of individual Web API client applications, which require extensive manual effort, starting with the search for suitable Web APIs, interpreting the documentation and proceeding to implementing custom implementation solutions. As already pointed out, even in the case of RESTful services, most of these tasks still require significant human involvement. Overall there is no established reference implementation of an invocation engine that supports the automated Web API invocation, given its description. Furthermore, dynamic invocation and exchanging or binding APIs at run time are still unaddressed, mainly because there is no solution for the underlying simpler tasks.

Supporting API invocation has been addressed in a number of previous research efforts, the main approaches probably being [AMG+10] and [LJ10]. SPICES [AMG+10] is a Web-based tool where end-users can interact with semantically annotated services, both WSDL and RESTful, and directly invoke them. In the context of Web APIs, the tool supports invocation of parameterised URIs over HTTP GET, where the parameter values are directly mapped based on the labels and there is no explicit lowering. Therefore, SPICES provides a limited invocation

[12] `http://code.google.com/apis/discovery/`
[13] `http://developer.mashery.com/iodocs`

solution, due to the insufficient description capacities of the used model. With similar interests, the work in [LJ10] aims to enable the automated invocation of RESTful and RPC-style services. It presents an approach that draws the service interface into an HTTP ontology, and uses backward-chaining rules to translate between semantic service invocation instances and the HTTP messages passed to and from the service. This approach discards the widely recognised lowering and lifting mechanisms and works solely at the ontology level.

In summary, the main limitations of current invocation solutions and approaches are based on the fact that they are only able to support a certain type of Web APIs, such as strictly RESTful ones, or the ones conforming to the particular platform requirements, which represent only a very small subset in most cases. Automated invocation based on the meaning of the inputs and the outputs, and the aim to achieve a certain goal is only partially supported by two existing approaches. Furthermore, the dynamic discovery and invocation of Web APIs, which is already well supported in the context of WSDL-based services, is still out of the scope of current research work. Furthermore, research on Web API invocation commonly disregards the need of handling authentication before the actual invocation can be performed. Therefore, in the following section we provide an overview of frequently used authentication approaches.

4.2 Web API Authentication

In this section we describe existing approaches for handling security and, in particular, authentication in the context of Web services and Web APIs. Authentication plays a very important role in supporting the use of Web APIs through automated invocation approaches, since more than 80% of the APIs reviewed required some form of authentication [MPD+10b] (see Chapter 6).

4.2.1 WS-Security

The issues of authentication and security have already been tackled in the context of WSDL and SOAP-based Web services. The result is a unified Web service security standard. WS-Security (WSS) [NKMHB06] specifies a set of feature extensions to SOAP messaging, in order to provide message integrity and confidentiality. In addition, it also provides a mechanism for associating security tokens with message content and allows for a variety of signature formats and encryption algorithms. As a result, the defined enhancements provide support for ensuring that the message is not altered by a third party (message integrity), that its content cannot be read by anyone but the designated client or server (confidentiality) and that the user has the necessary credentials in order to access particular resources (authentication). This is realised by a set of associated specifications including WS-Secure Conversation, WS-Federation, WS-Authorization, WS-Policy, WS-Trust and WS-Privacy.

WS-Security provides the general guidelines and mechanisms for realising secure solutions, which can be used in conjunction with other Web service extensions and higher-level application-specific protocols to accommodate a wide variety of security models and security technologies. However, the particular implementation is only as secure as the developer designs it to be.

WS-Security addresses the main security issues in the context of Web services. However, in contrast to WSDL-based services, Web APIs are proliferating autonomously, without conforming to standards and independently from Web services. The result is a very heterogeneous set of practices and techniques, where security issues such as confidentiality and message integrity, guaranteed by the WS-Security standard, are not considered as crucial. In fact, as our authentication-specific study shows (see Section 6.3.7), security in the context of Web APIs is reduced only to authentication, which in turn serves mainly the purposes of access control, where providers want to restrict and track the number of requests, instead of providing data integrity.

4.2.2 Common Authentication Approaches

This section describes common types of Web API authentication approaches. Currently, most Web APIs use one of the following authentication mechanisms. We differentiate between approaches based only on authentication credentials (API Key or username and password), approaches using a transmission security protocol (HTTP Basic Authentication, HTTP Digest Authentication and OAuth), and approaches that use different parts of the HTTP request in order to transmit the authentication information. We start by describing authentication mechanisms relying only on the input credentials.

API Key. Currently, the most common way of Web API authentication is via an API Key (also called "developer key", "developer token", "token Id", "user Id", "user key"). Web APIs using this mechanism include Last.fm[14], New York Times[15] and Remember the Milk[16]. This authentication mechanism does not have any security measures for the message integrity and confidentiality but is instead only based on the necessary credentials. The user only needs to provide the API key, which can be obtained by signing up for the particular Web API. The key is transmitted either as a parameter, as part of the Web API URL, or directly in the HTTP request.

Each client that provides a valid API key is permitted access to the requested resources. This approach is very simple to use and to implement. However, since the API key is not protected in any way during the message transmission, but is rather sent directly as plain text, this method is

[14] `http://www.last.fm/api`
[15] `http://developer.nytimes.com/docs/best_sellers_api`
[16] `http://www.rememberthemilk.com/services/api/`

suitable for Web API providers, who only want to control the access to the available resources. For use cases where the user identity has to be verified, in order to retrieve protected data, a more secure authentication mechanism is required.

Username and Password. Similarly, to authentication via API key, authentication via username and password is also only based on the required credentials. It provides no message encryption or signature and the login details are transmitted as parameters of the request URI or are included in the HTTP request. Example Web APIs include Happenr[17] and FileSocial[18]. The user only needs to create an account for the particular Web API and can use the username and password (in some cases email and password, telephone number and pin, username and token or API key and private key) to access resources. Similarly to the authentication via API key, this approach is only suitable for providers who want to restrict the traffic and the number of requests to their APIs.

The first two authentication mechanisms are only based on the required credentials, while the following approaches, including HTTP Basic Authentication and HTTP Digest Authentication, are transmission security protocols. These, provide a higher level of security for the login details and the client's message.

HTTP Basic Authentication. HTTP Basic Authentication [FHBH99] provides a simple way for user authentication. It is based on a challenge-response model, where the HTTP server requests and validates the authentication of the Web client. Example Web APIs include Assembla[19] and Basecamp[20]. In order to access a Web page or a Web API operation, which requires authentication, the client needs to provide the corresponding username and password in the form of an authentication header (with value *base64encode* of the string *username+":"+password* [FB96]). The base64-encoded string is transmitted and decoded by the receiver, resulting in the colon-separated user name and password string, which are checked against the expected values. If the client does not provide any credentials, the server will respond with 401 Authorization Required [FGM+99] HTTP response code.

This type of authentication is very simple and is supported by all popular Web browsers. However, although it uses base64 encoding, it does no encryption and the username and password can directly be decoded from the transmitted message. Therefore, this type of authentication is only suitable for Web APIs with low data security demands.

HTTP Digest Authentication [FHBH99] follows the same process as the HTTP Basic one – request, credentials challenge and response. However, it only transmits a digest of the username

[17] `http://www.happenr.com/webservices/`, for example `http://happenr.com/webservices/getEvents.php?username=xxx&password=xxx&town=London`

[18] `http://filesocial.com/api/docs`

[19] `https://www.assembla.com/wiki/show/breakoutdocs/Assembla_REST_API`

[20] `http://developer.37signals.com/basecamp/`

and password, which cannot be directly decoded. Example Web APIs include Talis[21] and AdSpeed[22]. The first time a client sends a request to the server, the server responds with a *nonce* (a random string) and the *realm* (typically a description of the computer or system being accessed). The client uses the username and password, to compute the digest response (result of *MD5(username:realm:password)*) and the digest of the nonce (usually by using MD5 [Riv10]), which are put in the response. The server processes the response by retrieving the stored password for the user and testing the nonce. If the nonce is correct, the response digest is checked by using the nonce, username and password to compute a digest and compare it to the received one. If the two digests match, the client is permitted access to the resources.

Since this type of authentication is based on a digest of a combined set of values, it should be difficult to determine the original input when only the output is known. However, if the password itself is too simple, there is the option to do brute-force attacks, which involves testing all possible inputs and find a matching output. Therefore, HTTP Digest Authentication adds some additional security in comparison to HTTP Basic authentication, but is still vulnerable to password guessing and man-in-the-middle attacks.

OAuth. One protocol for making authenticated HTTP requests by using a token is OAuth [MA10]. It enables users to share private resources stored on one website with another site by using a token, which is an identifier denoting an access grant with specific scope, duration, and other attributes, instead of the username and password. Therefore, OAuth supports the interoperability and the combining of resources coming from different websites, in a way that is transparent for the user. Example Web APIs include Fire Eagle[23] and Delicious[24].

Whenever a Web API (or a website) needs to access resources from another Web API, the user is asked to provide his/her access information for the host Web API, while in the background, OAuth creates a token, which can be used by other APIs to gain access to the resources. As a result the username and password are kept private and unavailable for third-party websites and APIs, while the interoperability is still facilitated. This authentication approach is extremely important in the context of mashups, since it does not require that the user provides credentials for every Web API included in the composition, but rather relies on token-based user-transparent handling of authentication.

A more comprehensive solution is presented by Rosenberg et al. [RKD+09] who offer a semantic-based approach for providing authentication support for mashups, covering a variety of authentication mechanism (e.g., basic authentication, custom application IDs, OAuth, etc.). The authors

[21] `http://n2.talis.com/wiki/Platform_API`

[22] `http://www.adspeed.com/Knowledges/830/AdSpeed_API/AdSpeed_API_Overview.html`, for example `http://api.adspeed.com/?method=METHODNAMEparam1=VALUEparam2=VALUEmd5=SIGNATURE`

[23] `http://fireeagle.yahoo.net/developer/documentation`

[24] `http://delicious.com/help/api`

address the problem that currently mashups that combine different resources from at least one enterprise source, lack the ability to enable security, based on diverse security requirements in terms of authentication. The provided solution incorporates both a model and semantics for integrating security concerns into mashups, as well as a supporting implementation in the form of a Secure Authentication Services (SAS). One limitation of the approach is that it currently supports only HTTP basic authentication and OAuth. Furthermore, the authentication credential descriptions have to be created manually, in an XML-based BPEL like language, which might present a challenge. It is also not clear how many Web APIs have already been implemented by using the SAS.

So far we have described authentication based on different credentials and on using different authentication mechanisms. In addition, there are also two main ways of sending the authentication information to the Web API. One very common way is to directly provide the API key or username and password as parameters in the request URI. For example Last.fm[25] and Fire Eagle[26] use this approach. Since the authentication credentials are not protected in anyway, this way of sending data is suitable only for openly available resources, where providers want to control the access to the Web API but are not really concerned with enforcing access rights and user identity. The other common way of transmitting authentication credentials is directly in the HTTP request. This method is somewhat more complex because the client needs to construct the request, instead of only calling a parameterised URI. However, it enables a higher level of security since the information can be encrypted and signed. For example, this way of sending information is commonly used by the HTTP Digest authentication approach.

The most popular authentication approaches are based on the API key, or passing a username and password as parameters in the URI. However, HTTP Basic is also a commonly used approach and, therefore, a solution that provides support for passing credentials, alongside input parameters, would not provide a wide coverage. In summary, current authentication approaches have three main characteristics: 1) the required credentials, 2) the used authentication protocol, and 3) the way of sending the authentication information. These characteristics are taken into consideration when developing the Web API Authentication (WAA) Model, presented in Chapter 9.

4.2.3 Further Authentication Mechanisms

In this section we describe further existing Web API authentication mechanisms and solutions, which address different challenges in the context of authentication but have not yet reached greater popularity.

[25] `http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&artist=Cher&apikey=XXX`

[26] `https://fireeagle.yahooapis.com/api/0.1/`

OpenID. This approach targets to solve the problem of one single user being forced to have many different Web application and API accounts, in order to be able to execute a mashup. It is a method based on using a single login at a trusted provider to automatically gain access to other websites. In this way the user can log on to different services with the same digital identity, where these services trust the authentication body. Web site providers, which use OpenID[27] include AOL, IBM, Microsoft and others. OpenID is often seen as a complementary approach to OAuth, where OpenID credentials can be used as a basis for generating OAuth tokens.

WebID. This approach [SIS+11] is based on providing means for uniquely identifying a person, company, organisation, or other agent using a URI, by applying the same process used in OpenID. It was originally introduced as FOAF+SSL [SHJJ09] for RESTful authentication. The main approach is to enable the linking of the user agent (browser) to an URI, in order to restrict the access to resources only to members of a group or selected individuals. WebID[28] is an authentication protocol which uses X.509 certificates to associate a browser to a person identified via a URI. It also enables automated session login, in addition to interactive session login. Therefore, it requires the user to enter neither a password nor an identifier but rather uses SSL and a custom trust protocol. All transmitted data is encrypted and guaranteed to only be received by the person or organisation that was intended to receive it. WebID presents a novel authentication approach, which includes Semantic Web fundamentals in the authentication process and is supported by leading members in the field[29]. Still it remains to be seen if it will manage to establish itself as one of the most commonly used solutions.

Web-key. The solution offered by Web-key [Clo08] is quite similar to WebID. It provides an authentication mechanism, especially designed to tackle the difficulties arising in the context of mashup authentication. Web-key is an HTTPS URL convention for representing a transferable permission in a Web application. It binds each permission issued from the Web application to a randomly generated string (key), which is transmitted in the fragment segment of the URL (for example, `https://www.emaple.com/ app/#mhbqcmmva5ja3`). The keys are generated on behalf of the user for every Web API that is part of the composition. In this way, each Web API has its unique key, instead of the user having to share his username and password across all composite APIs and the complete authentication process is user-transparent. However, this approach is fairly new and there are only very few Web APIs, which have adopted it.

XAuth. Another authentication mechanism is XAuth[30]. It provides an approach for extending authenticated user services throughout the Web by issuing user browser tokens for each of the

[27] `http://openid.net`

[28] `http://www.w3.org/wiki/WebID`

[29] Give yourself a URI - on Tim Berners-Lee's blog, `http://dig.csail.mit.edu/breadcrumbs/node/71`

[30] `http://xauth.org`

participating services. In this way the provider can recognise, which users are logged on the services and not only give access to resources but also give additional relevant options. A different approach is followed by **Yadis**[31], which instead of suggesting a new authentication mechanism proposes means for automatically detecting, which authentication protocol a particular system is more likely to use. Therefore, Yadis addresses the question of how do we know, which authentication mechanism needs to be used, by providing a service discovery system that determines automatically, without end-user intervention, the most appropriate protocol to use.

In addition to the here listed authentication mechanisms, there is also one approach that uses semantic Web service descriptions in order to capture authorisation and privacy service properties [KPS+04]. The authors suggest that privacy and authentication policies should be incorporated into the OWL-S Web service descriptions. This additional information can then be integrated into the service matchmaking process, in order to be able to select only services with particular security characteristics, for example, only services that provide encryption. However, it should be pointed out that this approach is suitable only for WSDL-based services annotated in OWL-S.

4.3 Summary

Invocation is a crucial task in the context of Web API use. Currently, it is usually enabled through the implementation of custom clients or, in rare cases, based on libraries offered by the providers. As a result, the produced solutions are not reusable and are rarely compatible with each other, which is strongly desirable if compositions and mashups are to be developed. Furthermore, the majority of the Web APIs require some form of authentication, which is most often disregarded in existing invocation approaches. Therefore, this chapter provides an overview of research work and solutions in the context of SWS and Web API invocation. It also describes common credential types, the ways of transmitting them and the used authentication protocols.

Based on the analysis of current approaches and solutions targeted at supporting Web API invocation, it can be deduced that there are two main ways of providing invocation support. The first one is based on enforcing a new technology standard or assuming the wide acceptance of one, such as in the case of RESTful services. Given such a foundation, different theoretical solutions and implementations can be developed. The drawbacks in this case are that it might be difficult to convince service providers to adjust their Web APIs to conform to the newly introduced guidelines (see Chapter 6) or that a relatively limited number of Web APIs actually employ a certain set of communication principles, for example, REST. As a result the coverage offered by this type of approaches is rather limited. Furthermore, this would only enable support for invocation on the syntactic level, not being able to process services based on the meanings of the inputs

[31] `http://yadis.org`

and outputs, and the envisioned functionality or goals. A shared technology standard, would still require that developers search manually for Web APIs, read through the documentation and implement client applications.

The second type of approach for facilitating Web API invocation support is based on developing frameworks that enable the automated use of APIs that are registered within the particular platform, i.e., implemented in such a way that conforms to the chosen invocation mechanisms. As a result, there are only few APIs that can be used within the framework but the provided supporting functionalities require very little or no developer involvement. Naturally, such solutions have a limited coverage but might be very useful in restricted domains or use cases that require only a small number of Web APIs in order to complete a certain task.

However, based on the analysis of current Web API invocation approaches, we argue that the lack of a general solution can be traced back to the heterogeneity of the Web API description forms but is also due to the fact that the information relevant for invocation is not captured in a way that enables automated processing. Therefore, we advocate an approach for creating semantic Web API descriptions that include all the details necessary for completing the invocation task, thus unifying the majority of the APIs under a common description model.

Considering common authentication approaches, it can be observed that providers prefer to use simple solutions to control the Web API access by requiring the provisioning of credentials, such as an API key or username and password, directly as part of the invocation URI. Furthermore, it is evident that currently there is not one established authentication approach but rather a broad range of available options. The frequent use of highly heterogeneous non-standardised authentication mechanisms is yet another limitation that needs to be circumvented in order to enable the automated Web API invocation.

In the following chapter we explore work related to tools and mechanisms for supporting the creation of semantic Web API descriptions. Since the manual creation of descriptions is effort and time consuming, we investigate existing tools that provide annotation functionalities and approaches for automatically completing some annotation tasks, such as the search for suitable ontologies or determining the type of functionality that the Web API exposes.

Chapter 5

Annotation Approaches and Tools

Semantic Web Services (SWS) aim to address the limitations of services on the Web by augmenting the service descriptions with semantic metadata, in order to achieve a higher level of automation for common tasks such as discovery, compositions and invocation. However, in the context of Web services in general and especially for Web APIs, the creation of semantic descriptions is a time consuming and resource intensive undertaking. Therefore, in this chapter we consider approaches and solutions targeted towards enabling the (semi-)automated creation of semantic Web API descriptions. In particular, we discuss approaches for the recommendation of semantic annotations for the data model, service classification, as well as for assisting intermediate, yet necessary steps such as finding suitable ontologies. In addition, existing tools for ontology visualisation and service annotation are also described.

In particular we divide the discussed approaches into two main groups – annotation methods, and visualisation and annotation tools. We analyse the related work in terms of the types of service, which are supported (WSDL-based or Web API), and the input/output of the approach or tool (WSDL file, ontologies, service description model).

5.1 Annotation Approaches

This section focuses on research work related to supporting the creation of semantic descriptions of Web services and Web APIs. In particular, it considers research approaches and implementation solutions in the areas of automatically acquiring semantic Web service descriptions and annotation recommendation.

5.1.1 Automated Acquisition of Semantic Web Service Descriptions

In the context of SWS, the task of supporting the creation of semantic descriptions has already been addressed by a range of research propositions. In particular, there are a number of developed approaches aiming to support the acquisition of semantic Web service descriptions. Paolucci et al. [PSSN03] offer a simple solution by addressing the problem of creating semantic metadata (in the form of OWL-S) from WSDL. The result is a syntactical transformation of the WSDL description, which contains no semantic information because WSDL contains no semantic information, in the first place. A challenge that is not addressed by this approach, is the mapping of WSDL-based service properties to classes or instances in a domain ontology. In addition, the difficulty of determining a suitable domain ontology for service annotation is also not discussed.

In contrast, Sabou et al. [SWGS05] tackle precisely the problem of creating domain ontologies that can be used for the annotation of a particular service. The authors use shallow natural language processing techniques to assist the user in creating an ontology based on software APIs. The solution is applied within the scope of two ontology building processes, in the context of two concrete research projects, revealing some of the major aspects necessary for building a Web service ontology.

Focusing on the Web service annotation task, Patil et al. [POSV04] apply graph similarity techniques to select a relevant domain ontology for a given WSDL file from a collection of ontologies. The presented approach involves also work on matching XML schemas to ontologies in the Web services domain. The authors use a combination of lexical and structural similarity measures, based on the assumption that the user's goal is not to annotate similar services with one common ontology, but rather to use different ontologies. Therefore, they also address the problem of choosing the right domain ontology within a set of ontologies.

Finally, Hess and Kushmerick [HK04] employ Naive Bayes [ELM03] and SVM [TK01] machine learning methods to classify WSDL files into manually defined task hierarchies. In addition, there are a number of approaches, which use existing Web service repositories, in particular UDDI, and enhance them with rich semantic markup [AGDR03], [LMK05].

Currently there are two approaches especially developed in the context of creating Web API descriptions. Saquicela et al. [SBC10] present a semi-automatic approach for annotating Web APIs in the geospatial domain. In particular, work focuses on creating syntactic descriptions of the Web APIs and semantically enriching their parameters, through the partial automation of the process [SBC11]. The authors introduce a service description model based on method[1], input and output, as well as invocation, input value and output value. As a result, the description

[1]The method is used to denote the actual Web API operation, and not the HTTP method as the names suggests.

contains both the service properties, as well as the values that would be required for the invocation. Semantic information from DBpedia[2] and GeoNames[3] is used to create annotations for the service. The result is a set of Web API annotations, which are linked to the API and stored in a repository, in a proprietary format.

It needs to be pointed out that the creation of the descriptions is only possible based on a given working invocation example for the particular Web API. This represents a drawback, since in most cases invocation examples are hard to find and have to be prepared manually. Furthermore, the invocation system is restricted to handling only Web APIs that can be called over HTTP GET, providing no support for POST or DELETE requests.

The second approach is developed by Taheriyan et al. [TKSA12], who present a system supporting the creation of semantic models of services in a semi-automated way. The user has to provide examples of the Web API request URLs, based on which the system automatically proposes a service model that represents the semantics of the API functionality. Similarly to the previously introduced approach, this represents a limitation since suitable and actually invocable URLs have to be found and parameterised, with appropriate values in place. The resulting model can be refined by the user via a user interface. The system also produces the required lifting and lowering transformations. The complete models are stored in a local repository.

Approach	**Type of Service**	**Service Model**	**Input**	**Output**
Paolucci et al.	WSDL/SOAP	WSDL/OWL-S	WSDL	OWL-S (only syntactic info)
Sabou et al.	WSDL/SOAP	WSDL/OWL-S	WSDL	OWL-S & learned domain ontologies
Patil et al.	WSDL/SOAP	WSDL	WSDL	SAWSDL
Hess and Kush.	WSDL/SOAP	WSDL	WSDL	generic classification i.e. OWL-S
Saquicela et al.	op.-based Web APIs	new description model	sample set of Web APIs	sem. annotations, proprietary format
Taheriyan et al.	op.-based Web APIs	(KARMA + SWRL)	ex. request URLs	sem. annotations, proprietary format

TABLE 5.1: Annotation Approaches

Table 5.1 summarises the approaches for creating semantic Web service and Web API descriptions. The approaches that require a WSDL file, rely on the availability of syntactical information. Therefore, they cannot be directly applied on Web APIs and some modifications are necessary in order to enable the creation of semantic annotations on top of existing HTML documentation. In the context of Web APIs, solutions are available only for operation-based APIs and the result is a set of annotations, which are in a format specific to the particular approach. Therefore, the produced semantic descriptions are not compatible with each other and

[2]The DBpedia Ontology, `http://dbpedia.org/Ontology`
[3]GeoNames Ontology, `http://www.geonames.org/ontology/documentation.html`

can hardly be reused with other solutions. Furthermore, Saquicela et al. propose a solution only for the geospatial domain, while the approach of Ambite et al. is restricted to the annotations already available in the database.

5.1.2 Annotation Recommendation

In the context of automating the process of creating Web API descriptions, we consider recommendation approaches that could be adopted in order to directly suggest suitable annotations for the different parts of the API, including operations, inputs and outputs, but also for the API as a whole. In general, the main aim of recommender systems is to assist users in making decisions among different alternatives, based on user preferences. Or more precisely, recommender systems support users by identifying interesting products and services in situations where the number and complexity of offers outstrips the user's capability to survey them and reach a decision [TH01]. Therefore, in the context of semantic Web service descriptions, recommender systems can help a user select suitable annotations by suggesting only a subset out of a large collection of ontologies.

There are a number of recommender systems developed for the purpose of aiding users in providing semantic information and these systems are particularly relevant to our work on creating semantic descriptions of Web APIs. TagAssist [SH07] is a system, which provides tag suggestions for new blog posts by utilising existing tagged posts. Similarly, [LC07] use collective intelligence extracted from collaborative tagging, in addition to word semantics, in order to learn the best set of tags to use for new blog entries. The approach presented in [JMH+07] is an adaptation of user-based collaborative recommendation and graph-based recommendation, which suggests tags for different resources. All of these approaches use common recommender techniques (e.g. collaborative filtering), for the purpose of making the process of adding semantic information on the Web easier, by suggesting tags and annotations.

In general, there are different formats and methods for creating semantic service descriptions, however, there are some main tasks, which very commonly need to be addressed. These include the classification of the service based on its functionality, the finding and selection of a data model annotation, and the annotation of the individual service properties. ASSAM [HJK04], a tool that assists a user in creating semantic metadata for Web Services, offers a solution for some of these issues. It automatically suggests semantic metadata based on two main machine learning algorithms. The first one, aims to semantically classify Web services, while the second one, facilitates recommendation of semantic annotations for the service inputs and outputs, by aggregating data returned by multiple semantically related Web services. A similar approach is taken in the METEOR-S Web Service Annotation Framework [POSV04], a framework for semi-automatically marking up Web service descriptions with ontologies. The framework focuses on

determining a relevant domain ontology for making data model annotations based on WSDL files. Classification is also applied by using classification domain ontologies to categorise Web services into domains.

METEOR-S addresses one of the main challenges of creating semantic descriptions of Web APIs, namely, determining suitable domain ontologies. Similarly to [SWGS05], some research [NV04, ZN08] on the semantic description of Web services focuses on learning domain ontologies for service annotation. Such approaches are effective because learnt ontologies are more service-specific and, therefore, more suitable for making annotations than general purpose ontologies. However, this means that each service or group of services has its own ontology and as a result, common tasks such as service discovery, composition and invocation, have to devote processing effort related to ontology mapping and concept matching, which would not be the case if a set of common ontologies were used for the annotation. In addition, instead of reusing annotations from already semantically described services, service-based ontology learning adds additional complexity to the service annotation process.

Recently, with the growing use and popularity of Linked Data, there is an abundance of semantic data available and Linked Data represents a very valuable source of ontologies and background knowledge. This provides the basis for enhancing existing techniques for annotation recommendation to use this semantic data [SBC10, TKSA12]. In addition, there are tools such as Watson [dSM+08] and Sindice [TDO07], which assist users in searching for semantic tags and ontologies. In the context of creating semantic service descriptions, Linked Data provides the basis for the development of annotation approaches, based on reusing published semantic data.

In contrast to most recommendation approaches, Billsus and Pazzani [PB07] address the problem in a different way. They suggest that the recommendation of alternatives based on user ratings can be solved by transforming the recommendation problem into a classification problem. Automatically assigning a Web service to a particular group of services with similar functionality or similar application domain, already provides semantic information, which directly contributes to the improved automation of the service discovery and composition tasks. Moreover, in some approaches, the classification task can be equivalent to determining a suitable domain ontology for the annotation of the service [HK03]. In other approaches classification can at least simplify the process of determining a domain ontology and finally, sometimes [OTSV04] annotation recommendations are computed as part of the classification process. Therefore, most approaches for acquiring Semantic Web Service annotations rely on service classification for both determining the type of functionality that is provided and selecting domain ontologies. Commonly used classification approaches are the k-nearest neighbour, naive Bayes and Rocchio [OTSV04] algorithms, while naive Bayes is the most commonly used one. Support vector machines (SVM) based on document frequency values are also used in Web service classification and annotation [BCPS05].

In summary, there are two systems that support the semantic annotation of WSDL-based services – ASSAM and the METEOR-S Web Service Annotation Framework. They both take as input the XML service description file and produce OWL-S and SAWSDL descriptions, correspondingly. In the context of adding metadata to Web APIs, Taheriyan et al. [TKSA12] and Saquicela et al. [SBC10] are the two main approaches that aim to leverage Linked Data in order to create semantic annotations. They take as input example requests and endpoint URIs, and return a set of annotations for the Web API inputs and outputs. Finally, approaches that facilitate classification can be used as a basis for determining suitable domain ontologies, which can be employed in order to describe the APIs.

5.2 Ontology Visualisation and Annotation Tools

In this section we provide an overview of general semantic data tools, as well as tools that are especially developed for supporting the creation of semantic Web service descriptions. In particular, we discuss ontology visualisation tools and semantic annotators. In the context of SWS, we look at WSMO Studio [DSK+07], the Web Service Modelling Toolkit [KMTF07] (WSMT) and ASSAM [HJK04]. It is important to point out that the work by Taheriyan et al. [TKSA12] and Saquicela et al. [SBC10] is also relevant in the area of annotating Web APIs, since the output of the two approaches is metadata related to the inputs and the outputs of the service. However, we only mention them here shorty, since it was already discussed in the previous section.

The process of semantic annotation can be assisted by ontology visualisation tools, which enable users to view and explore an ontology and to decide whether or not it can be used for the annotation of a particular service. A good overview of current ontology visualisation tools is provided in [KHL+07], including common tools with explorer-based view of the ontology such as Protégé [NFM00], OntoEdit [SAS02], NeOn Toolkit [HLSE08], Kaon [KAO02, BEH+02] and OntoRama [ERG02].

Ontology visualisation already provides some user support, however, functionalities such as adding or viewing annotations are even more relevant in the context of supporting the creation of semantic descriptions of Web APIs. There are tools, which enable the annotation of Web content such as OntoMat-Annotizer [HS02] and SMORE [KG05]. OntoMat-Annotizer is a webpage annotation tool, which supports users in creating and maintaining ontology-based OWL-markups. A similar tool is SMORE, which enables users to markup HTML documents in OWL by using Web ontologies. It includes some options for ontology editing and provides users with the possibility to create a new ontology based on terms from web documents. The results of both tools are a webpage with attached markup.

PowerMagpie [DMD07, dMD+08] is a tool that uses semantic information in a quite different way. PowerMagpie was initially implemented as a tool using ontologies to markup Web documents. Its current version has evolved into a semantically-enhanced Web browser. While browsing, PowerMagpie identifies and provides any available semantic markup, which can be found on the Web. It is not restricted to one predefined ontology, but rather accesses the whole of the semantic data on the Web through Watson [dM11] and selects and presents to the user relevant information. The approach used by PowerMagpie, lays the foundation for the development of tools that enable the semi-automatic creation of semantic Web API descriptions.

Until now we discussed tools that support ontology visualisation as well as tagging and annotation of web content. In the following, we describe applications that are especially developed for the creation of semantic Web service descriptions.

WSMO Studio [DSK+07] is developed for modelling and creating Semantic Web Services and semantic business processes. It provides a modelling and visualisation framework for assisting users working in the WSMO domain with tasks related to the semantic Web service annotation. Similarly, the Web Service Modelling Toolkit (WSMT) [KMTF07] comprises a number of tools for Semantic Web Services, based on WSMO, WSML and WSMX, including a Web Service Modelling Language visualiser, a WSML reasoner and a WSMX data mediation mapping tool.

SOWER[4] (also known as WSMO-Lite Editor) is an editor, which enables the manual annotation of WSDL service descriptions with semantic information, following the WSMO-Lite service ontology specification and using the SAWSDL annotation mechanism. Using the tool, the user is able to create new annotations on existing descriptions and also modify or remove already created annotations. The editing process itself is completely manual and is based on drag-and-drop, context and drop-down menus, thus hiding formalism complexities from the user.

Currently, the only tool that is particularly targeted at the semi-automated creation of semantic Web service descriptions is ASSAM [HJK04]. It includes a WSDL tree-based viewer, a category browser, as well as a datatypes display and a service browser. These components are common also for other annotation tools, however, the key feature of the WSDL annotator is its ability to suggest, which ontological concepts to use for annotating service elements. The result is a higher level of automation in comparison to other existing tools for creating semantic Web service descriptions.

[4] `http://technologies.kmi.open.ac.uk/soa4all-studio/provisioning-platform/sower/`

5.3 Summary

Providing tools and mechanisms for supporting the creation of semantic Web API descriptions, is an important part of enabling the declarative and formal description of service semantics. Since the manual creation of descriptions is effort and time consuming, we investigate existing tools that provide annotation functionalities and approaches for automatically completing some annotation tasks, such as the search for suitable ontologies or determining the type of functionality that Web APIs expose.

Based on the analysis of approaches and tools in the context of enabling the semi-automated creation of semantic Web API descriptions, we can conclude that currently there is no implemented framework or methodological solution that provides the required support for the majority of the Web APIs. Still, there are a number of solutions that can be adopted in order to reduce the amount of manual effort required for the completion of some annotation tasks. In particular this is true for determining the type of functionality that the service provides and for retrieving a set of semantic entities, which can be used to enhance individual service properties or the service as a whole. Since these research areas exhibit a plenitude of approaches and implementation solutions, it must me determined how this work can be adopted to address challenges in the context of semantic Web APIs.

Up to date, there are only two approaches, which to some extent support the semantic annotation of Web APIs. In particular, Taheriyan et al. [TKSA12] and Saquicela et al. [SBC10] take as input example requests and endpoint URIs, and return a set of annotations for the Web API inputs and outputs. These approaches show some limitations, since they are suitable for only a particular type of API and have been applied only to limited domains. Another possible solution can be based on existing tools for the annotation of WSDL-based services, which can be analysed and used for determining requirements and guidelines for the necessary computational and visualisation components. Approaches for ontology visualisation and browsing can be simplified and adopted from popular tools such as Protégé. Since the majority of the existing Web APIs are described in HTML webpages, a necessary feature for a tool developed to support the creation of semantic descriptions of Web APIs is to be able to directly manipulate the HTML content displayed in a Web browser, similarly to PowerMagpie.

In terms of developing an application that enables the semi-automated creation of semantic Web API descriptions, it is important to identify, which are the most time and effort consuming tasks, within the annotation process, and especially focus on providing solutions towards easing their completion. Similarly, the provisioning of semantic details, which are crucial for enabling certain service tasks, should be prioritised. This is especially true for describing the inputs and the outputs of the service as well as the functionality that it provides.

In addition, we aim to reduce the level of expertise required by the people creating semantic Web API descriptions, who can be service providers, developers or API users. Therefore, we aim to hide formalism complexities behind a user interface, requiring only the completion of simple tasks such as the selection of the text related to a particular service property and clicking on a corresponding element for making an annotation.

Part III

Supporting Open Services on the Web

Chapter 6

On the Current State of Service on the Web

Currently the world of services on the Web can be split into two main groups – the "traditional" Web services, based on WSDL and SOAP, and the Web APIs, also referred to as RESTful services [RR07] when conforming to the REST architectural principles [Fie00]. Recent years have been marked by the proliferation and increased use of Web APIs. However, despite their popularity, the use of Web APIs is still characterised by a number of challenges, mostly resulting from the fact that in contrast to Web service technologies, work around Web APIs has evolved in a rather autonomous way, conforming to no particular guidelines or standards, leading to a wide range of description formats and structures and different levels of description detail. Therefore, in this chapter we present an analysis of the current state of services on the Web, focusing especially on Web APIs. Traditional Web services are already largely studied and their characteristics are first of all predefined by specifications and standards, and second of all, can be surveyed through a number of established WS repositories [SMP10]. In contrast, the current state of Web APIs remains largely unexplored. Therefore, in order to be able to provide adequate approaches and a description model for addressing the challenges faced by Web APIs and supporting their use, we first need to gain a comprehensive overview of the current landscape of APIs on the Web, including their characteristics, existing correlations and trends.

The work presented in this chapter serves as a reality check over the current state of Web APIs. It provides a clear picture of current Web API properties and features, and lays the foundation for identifying deficiencies and developing approaches that support the more automated Web API use.

6.1 Introduction

Despite the fact that there are already a number of approaches targeted at describing Web APIs [W3C07a, Had06] and supporting their use by the means of semantic technologies [KV08, SGL07], none of the existing proposals are based on actual studies or data that reflects the current state of Web APIs. Before any significant impact and improvement can be made to current Web API practices and technologies, we need to reach a deeper understanding of how Web APIs are published and used. This involves, for instance, figuring out how current APIs are developed and exposed, what kind of descriptions are available, how they are represented, how rich these descriptions are, etc. We need to gain a clear picture of the current state and practices with Web APIs, in order to be able to identify deficiencies and realise how we can overcome existing limitations, how much of the available know-how can be applied and in which manner.

This chapter focuses on a thorough analysis over a body of publicly available API descriptions. We conduct two consecutive studies, the first one targeted at gathering an initial overview of API characteristics, the second one revisiting the gathered result and exploring some further properties in more details. In particular, we analyse how Web APIs are published, we check which information is provided and its level of detail. We investigate the characteristics of input parameters and record the type of functionality that the API provides. Similarly, we study the provided output descriptions and analyse the different types of APIs interface technology implementations, as well as the availability of relevant details such as the HTTP method, invocation URI and authentication requirements. We also record whether example requests and responses are provided, since they indicate how the communication between the client and the server is realised. Finally, we also study general API information, such as the number of mashups and operations, in order to be able to draw conclusions about the reuse and the granularity of the APIs.

This chapter is structured as follows: Section 6.2 provides an overview of current trends surrounding Web APIs. Section 6.3 and Section 6.4 describe in detail the two Web API studies that we conducted, including the setup, the individual characteristics that were analysed, the gathered results and a discussion of some identified trends and correlations[1]. Section B.2 introduces the survey system that was used to conduct the second study. It takes the form of a Web application, is highly configurable and gathers the data in RDF format. The chapter is concluded by a short summary.

[1] All the gathered data is available at `http://purl.oclc.org/NET/WebApiSurvey/`.

6.2 The Proliferation of Web APIs

Web APIs have been gaining in popularity and use during the past couple of years, becoming an important trend in the context of services on the Web. This is especially evident by comparing the development of the number of available traditional Web service and Web APIs over the past few years.

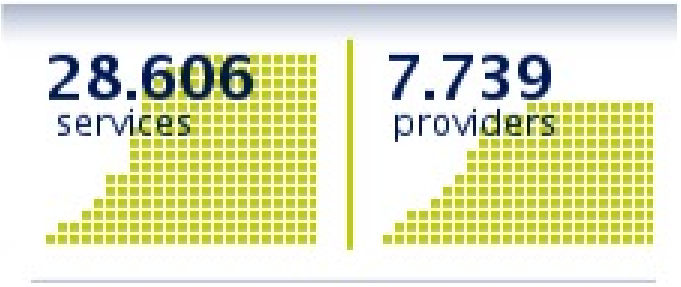


FIGURE 6.1: Web Services/Providers Timeline (Total Numbers from 2007 to 2012)

Figure 6.1[2] shows the number of currently existing Web services, as given by the Seekda WS search engine. The important aspect is that there is no significant growth or decline, and this picture has remained relatively unchanged during the past four years. After the initial growth in the number of WS, a level of stagnation was achieved during 2010. In contrast, the number of APIs registered with ProgrammableWeb is exponentially increasing, with more and more entries over a shorter period of time.

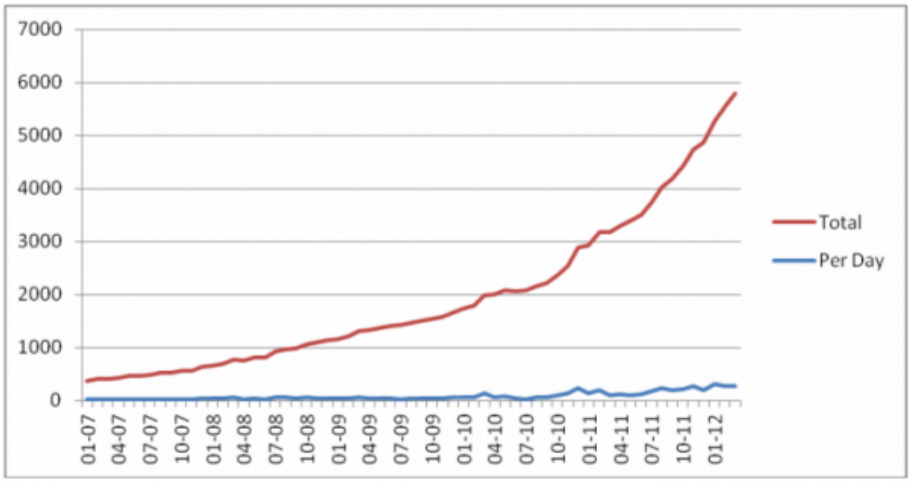


FIGURE 6.2: APIs Timeline (Total Numbers Quarterly from 2007 to 2012)

Figure 6.2[3] shows the growth-rate of the number of available Web APIs. As can be seen, there is a continuous growth, characterised by a significant increase during 2011. In fact, currently the number of registered APIs has reached over 11000[4]. This clearly demonstrates a strong increase

[2]Source – Seekda Web Services search engine, `http://www.seekda.com/`, last viewed April 2012.
[3]Source – ProgrammableWeb, `http://www.programmableweb.com`, last viewed January 2012.
[4]Last viewed March 2014.

in interest, which is also reflected on the developer side by the growing number of applications and mashups built on top of the APIs.

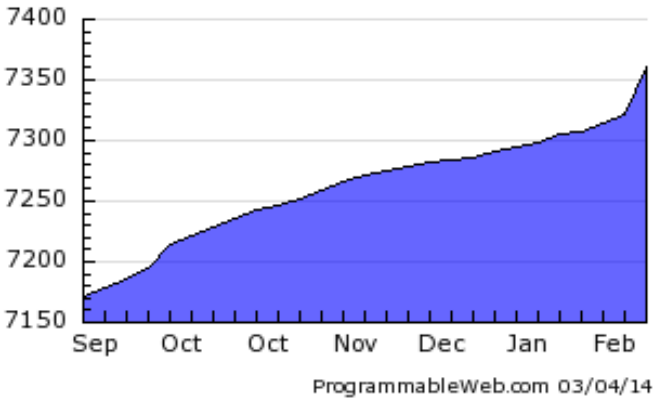


FIGURE 6.3: Mashups Timeline (Total Numbers from September 2013 to March 2014)

Figure 6.3[5] shows a constant increase in the number of mashups over the past six months. This illustrates that more and more developers are using the available Web APIs in order to create new solutions benefiting from the exposed resources.

Despite these trends in the context of Web API provisioning, up-to-date the current state of Web APIs, including different description forms, input types, invocation details, etc., has remained unexplored. However, there are two similar studies, devoted to investigating Web services on the Web.

The authors in [LLZ+07] provide a study on Web services, focusing on deriving statistics based on operations analysis, size analysis, word distribution and function diversity analysis by using the Google API. This study is based only on a few Web service characteristics and is restricted to only one source.

A broader and more complete study is given by [AMQ08]. The authors have developed a crawler for collecting metadata about service interfaces available through repositories, portals and search engines. The gathered data is used to determine statistics about object sizes, type of technology and functioning of the Web services, among others. In comparison to previous studies, this one also provides conclusions about the status of Web services and what percentage of the Web services are considered to be active and responsive.

It is important to point out that both studies cover only Web services and not Web APIs. Therefore, the analysis results presented here are the first ones that directly contribute to gaining a better understanding about the overall state of Web APIs.

[5] Source – ProgrammableWeb, `http://www.programmableweb.com`, last viewed March 2014.

6.3 First Web API Survey

In this section we describe the first Web API survey that was conducted in order to gather insights about common Web API characteristics.

6.3.1 Methodology

The first Web APIs survey was conducted during February 2010, manually analysing the description of 222 Web APIs from the ProgrammableWeb[6] directory. ProgrammableWeb is a popular API directory, that at the time of the study provided information about 2002 APIs and 4827 mashups. In order to enable basic search, the APIs are sorted in categories. The analysis covered all 51 categories, aiming to provide domain-independent results. The analysed Web APIs covered all categories by choosing the top 4 APIs within a category, as listed by ProgrammableWeb. However, since some categories have only one or two entries, we compensated this by taking additional APIs from the previous and following categories. Therefore, the analysed number of Web APIs per category varies. Overall, the survey covered 18% of the Web APIs that are marked as RESTful in ProgrammableWeb[7] (222 APIs out of 1235 APIs, at the time of the study). Therefore, we consider the following results to be representative for the directory and in general, since ProgrammableWeb is currently the largest directory[8].

Each Web API description was analysed in terms of six main groups of features, including general Web API information, type of Web API, input parameters, output formats, invocation details and complementary documentation. The Web API analysis was conducted manually, and some features such as the *type of Web API* were examined twice in order to achieve greater accuracy. More concretely, each Web API was examined in terms of:

1. **General Web API information** – we recorded the name of the API, its description, category, number of mashups, date updated, URL and number of operations.
2. **Type of Web API** – we checked whether the API description is RESTful, RPC-style or hybrid (for more details see Section 6.3.3).
3. **Input parameters** – we collected details on the use of default parameters, optional parameters, coded parameters (for example, instead of "English" use "en"), parameters with alternative values (for example, the input value is 1 or 2 or 3), and whether the datatype of the input parameter is stated and boolean (yes/no, true/false) parameters are used.

[6] `http://www.programmableweb.com`

[7] ProgrammableWeb classifies APIs according to the used protocol/style into Atom, Blogger, JavaScript, iCal, REST, SOAP and XML-RPC.

[8] Webmashup.com (`http://www.webmashup.com`) contains around 1800 Web APIs and 3100 mashups, while APIFinder (`http://www.apifinder.com`) provides around 1100 Web APIs, numbers from February 2010

4. **Output formats** – we recorded the form of the output (for example, XML or JSON) and whether it is determined via an input parameter.

5. **Invocation details** – we checked whether the HTTP method and the invocation URI are provided. Furthermore, we recorded whether the API requires authentication and if yes, what type, how are the input parameters transmitted and how is the authentication information transmitted.

6. **Complementary documentation** – we examined the provisioning of example request, example response and a list of error messages/codes.

We focus our analysis on studying precisely these groups of API features because each of them plays an important role for different aspects of the API use. The general information provides insights on the characteristics that are commonly used to describe Web APIs and how this information is captured, including reusability and level of granularity. Since, an important part of current research work on APIs is focused on investigating and comparing different Web service types (REST vs. WSDL and SOAP) [Pau09], we also record and analyse the existing types of Web APIs. We study input parameters, output formats and invocation details, since they serve as the basis for conducting main service tasks, focusing on invocation and authentication. These Web API features are present in all interface description languages (IDLs), as they are considered essential for invocation [Pau09], composition [GRN+08] and discovery. The complementary documentation provides details on how the communication between the client and the service is realised, and what are the possible errors that can occur.

The analysis approach involved a sequence of steps. First, for each API picked for the study, the ProgrammableWeb[9] webpage was opened. The APIs to analyse were randomly chosen within each category, covering all categories. This was necessary in order to ensure that the results are domain-independent and at the same time representative for the whole directory (see Section 6.3.2 for more details). For each API the general information was recorded. Second, the provider's Web API documentation was examined, recording the documentation URL, counting the number of operations and determining the type of the API interface (for more details see Section 6.3.3)). We also analysed the input parameters of each operation, in case of RESTful services these are also referred to as the *scope* [RR07]. For the output of each API, the serialisation format was recorded, including the available alternatives and how they are chosen (through parameterisation, through a separate URI for the invocation, or through content negotiation). Finally, the invocation details, included in the description, and the complementary documentation were recorded. Our objective was to gain a picture of the current state of the Web APIs landscape as depicted by their documentation. We did not perform any test invocations of the APIs, which would be necessary in order to identify discrepancies between the actual implementation and the provided documentation.

[9] `http://www.programmableweb.com`

The documentation of every Web API had to be reviewed manually. In the process, we already noticed that the work was slowed down by the fact that the description forms and structures are very diverse and each API had to be examined from scratch, without being able to benefit from the analysis of previous APIs. This already provides some indication about the difficulties arising from having to deal with heterogeneous textual API documentation.

In the following sections we present the results of our study. The results are structured into six groups, according to the different parts of the API descriptions that were analysed. Each group provides valuable insights about separate aspects of the APIs and serves as the basis for identifying common characteristics and drawing conclusions.

6.3.2 General Web API Information

The analysis of the general Web API information includes the recording of some details provided directly by the API directory, such as the name of the API, its description, the category that it is assigned to, the URI of the API and the latest update of the description. Table 6.1 provides the numbers for these features.

Description	Maximum	Minimum	Average
APIs per Category	12	1	4
Number of Mashups	506	0	6.4
Number of Operations	over 200	1	15.5

TABLE 6.1: Survey 1 - General Web API Information

Of these general details, the number of mashups is of particular relevance because it provides a measure for the level of reuse of Web APIs and to a certain extent can help to highlight factors influencing the reusability of APIs. Since we cannot collect data about the actual API use, which is available only to providers, the number of mashups[10] is the only measure indicative for the popularity of an API. The analysis shows that a few APIs are highly reused, whereas most APIs are used in very few or no mashups at all. In particular, there are 136 APIs with 0 mashups, 60 APIs with 1 to 4 mashups and 26 APIs with 5 to 506 mashups (see Figure 6.4). The API with most mashups is Flickr[11], which can be easily integrated into different Web applications as a source of images and photos. In summary, there is a big difference in the frequency of reuse of some APIs, while most APIs are not used often as part of mashups. Also it must be noted, that the number of mashups is as provided by ProgrammableWeb, therefore the actual values can somewhat differ. However, for comparison purposes it is still representative, since the data comes from the same source for all APIs, thus being affected by the same inaccuracy.

[10] As provided by ProgrammableWeb.

[11] `http://www.flickr.com/services/api/`

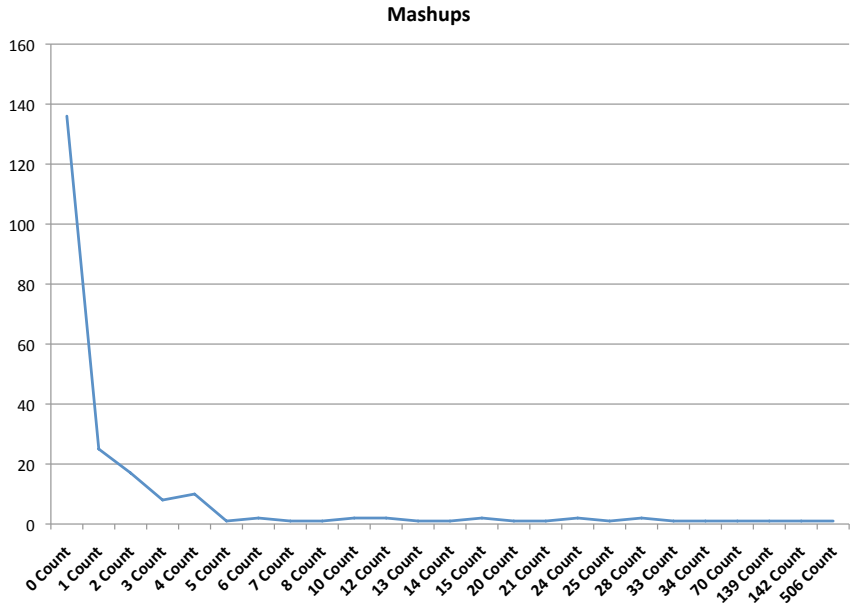


FIGURE 6.4: Number of Mashups (APIs per Number of Mashups)

The collected general API information also delivers some valuable insights about the granularity, i.e., the number of operations, of the APIs. 109 of the APIs or about 50% have 1 to 7 operations, while 36 APIs or 16% have only 1 operation. 92 APIs have between 7 and 50 operations, where more APIs have fewer operations. Finally, 21 APIs have between 50 and 200+ operations (Yahoo Ads[12]) (see Figure 6.5). This leads us to the conclusion that the majority of the APIs are small and have very few operations. We investigated whether there is a correlation between the size of the APIs and their use as part of mashups, but even though social and community websites, seem to expose a larger number of operations, there are important exceptions such as del.icio.us[13], which has only 15 operations but 142 mashups, and geocoder[14] with 3 operations but 28 mashups.

Finally, we discuss two characteristics that do not relate so much to the individual Web API descriptions but rather to the way of storing details in the ProgrammableWeb directory. First of all, our analysis highlighted that since all details are added manually to the Web API directory, some of the details were not always accurate. This is especially true for the URL of the documentation, which had sometimes been moved or was no longer available, and for the authentication information, which was very often inaccurate. In this case, we used Google to search for the website with the API documentation and referred to the provider's information for determining the correct authentication method. Second, some of the entries were outdated, which was especially obvious when the Web APIs no longer existed. During our first survey, we

[12] `http://developer.yahoo.com/everything.html`
[13] `http://delicious.com/help/api`
[14] `http://geocoder.us/help/`

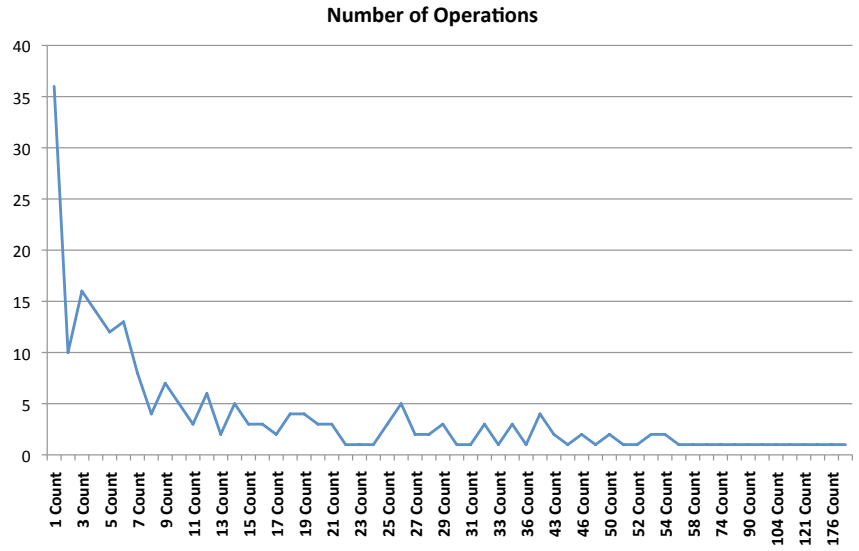


FIGURE 6.5: Number of Operations (APIs per Number of Operations)

did not record how many APIs were in the directory but no longer available and simply chose another API. In the second survey, we collected this information (see Section 6.4).

This is indicative for the difficulties resulting from using directories based on manual entries. First, there is the possibility to retrieve outdated information, because the entries cannot be automatically updated. Second, the retrieved information can be erroneous, due to wrong or inaccurate user input. Therefore, despite the fact that currently these manually created directories are the easiest way to search for APIs, since the included general information is often inaccurate, it is impractical to develop approaches that are based on more detailed properties, such as the ones required for supporting discovery, composition and invocation.

6.3.3 Type of Web APIs

In this section we describe our findings regarding the different types of Web APIs and their frequency of use. We have identified three types of APIs: RESTful, RPC-style and Hybrid. RESTful services are defined as services, which conform to the representational state transfer (REST) paradigm [Fie00]. REST is based on a set of constraints such as the client-server based communication, statelessness of the request and the use of a uniform interface (see Section 3.2). A RESTful service comprises a collection of uniquely identified resources and their links to each other, and is commonly implemented by using HTTP. In addition, RESTful services are characterised by resource-representation decoupling, so that resource content can be accessed in different formats.

It is important to point out that for the scope of our study, we identify Web APIs as RESTful, when their descriptions indicate that they are resource-centred and data retrieval and manipulation is done only over the HTTP methods (for a complete list of the RESTful service requirements see Section 3.2). We define RESTful APIs in this way, for the scope of our survey, because REST principles dictate the architectural style of the application and are applied to the actual implementation. Therefore, what we can observe in the Web API documentation is only a propagation of the underlying technical realisation. For example, we cannot determine from the documentation whether an API is based on a layered system and if the responses support caching mechanisms. As a result, we classify the APIs based on the appearance of the interface, as captured in the documentation. Example APIs, which were classified as RESTful, include MusicBrainz[15] and Doodle[16]. RESTful APIs can have a scope, or a set of parameters, to restrict the effect of the HTTP methods on the resource only to the ones determined by the parameter values. For example, instead of retrieving all news resources in the *News* collection by using GET (HTTP GET http://url/.../News), the API can also be invoked by including a parameter and retrieve only news created by a particular user (HTTP GET http://url/.../News?user=aUser).

Furthermore, we define RESTful APIs based on these two criteria because they determine how the actual communication with the service is realised, which is key for supporting invocation. In other words, if we know that an APIs is resource-centred and uses directly HTTP-based messaging and communication, it is clear how the HTTP request and responses can be processed. For this reason, we are not more restrictive, including requirements such as determining the output format only through content negotiation and realising state changes only through links (HATEOAS). This would enable us to identify what percentage of the Web APIs are strictly-REST conforming (observed in the second survey, see Section 6.4), however, it would not support us in developing a Web API description model, covering the majority of the APIs.

We define RPC-style APIs as APIs that are not based on resources but rather on operations, which can perform simple actions, such as read or write, or more complex ones, such as the computation of functions or a multi-step process. These operations are then invoked over HTTP, where the semantics of each operation should conform to the semantics of the used HTTP method (for example, an update of a record is realised using HTTP POST and not HTTP GET). For example, an RPC-style API, providing the same information as the news RESTful one, would look like: HTTP GET http://url/.../getNews and there can be a scope or a set of parameters (HTTP GET http://url/.../getNews?user=aUser). Example APIs include GeoNames[17] and Daylife[18]. It is important to point out that, similarly to RESTful APIs, we base our classification strictly on the API documentation, since we have no direct information about

[15] `http://wiki.musicbrainz.org/XMLWebService`
[16] `http://doodle.com/xsd1/RESTfulDoodle.pdf`
[17] `http://www.geonames.org/export/web-services.html`
[18] `http://developer.daylife.com/docs`

the underlying implementation. Still our definitions of Web API types share a common understanding with the ones given in [RR07], stating in essence that RPC APIs expose internal functionalities through an arbitrary programming-language-like interface that is different for every service, while resource-oriented APIs expose internal data through a simple uniform document-processing interface.

Hybrid APIs, as the name suggests, represent a mix between RESTful and RPC ones. Hybrid-style APIs define their own operations, but employ operation information, which is contradictory to the used HTTP method. For example, a hybrid API can realise the *getNews* operation through POST and *addNews* through GET. Example hybrid APIs include ClearForest (http://www.opencalais.com/documentation/calais-web-service-api), which uses POST for getting resources and Box.net (http://developers.box.net/ApiOverview) where adding a new element can be done by using GET. The use of hybrid APIs can be very problematic since they do not guarantee operation safety, especially in cases where data manipulation is realised by using GET, because of the possibility of unintentional data modification. In such cases a simple crawler can change or delete resources, since it would use GET, expecting to retrieve information instead of altering it.

Description	In %
RPC-Style	47.8%
RESTful	32.4%
Hybrid	19.8%
Mashups with RPC-Style APIs	42%
Mashups with RESTful APIs	34%
Mashups with Hybrid APIs	24%

TABLE 6.2: Survey 1 - Type of Web APIs

Table 6.2 shows the distribution of the different types of APIs. Almost half of the Web APIs are RPC-style and about one third are RESTful. The hybrid APIs represent about 20% of the analysed data. This shows that, even though RESTful services are by design based on the same principles as the Web, their level of adoption is still relatively low. Instead of identifying resource collections and manipulating them with the help of HTTP methods, developers prefer to define their own operations, whose functionality sometimes even contradicts the used HTTP method (hybrid APIs). As a result, two thirds of the API descriptions are based on operations and not resources, disregarding the REST principles.

A very similar distribution can be detected among the APIs, which are reused as part of mashups. 42% of the APIs are RPC-style, 34%– RESTful and 24%– hybrid[19]. Therefore, we can conclude that API reuse is not driven by the type of description, since the mashups percentage distribution

[19]The percentages are determined by dividing the total number of mashups to the number of mashups that are realised using RPC-style, RESTful and hybrid APIs, correspondingly.

matches almost exactly the Web API distribution. As a result, contrary to common belief, we can argue that the current proliferation of Web APIs cannot be attributed to the use of RESTful services. As our study shows, most Web APIs do not have RESTful descriptions and how APIs are described does not seem to have an impact on their reuse.

6.3.4 Input Details

In this section we present an analysis of the information in the API documentation, relating to the input parameters. As can be seen in Table 6.3 about 60% of APIs use optional parameters, while 45% use default values. This has a strong effect on the matchmaking and invocation approaches, since one API can be found or not depending on whether optional parameters are taken into account or not. Similarly, if invocation is done on the basis of default values, the output results can be drastically changed. For example, a lot of APIs have XML as a default output format but some use also JSON as default. If the default parameter value is used, the results might be retrieved in the wrong format.

Description	Number	In %
APIs with optional parameters	136	61.3%
APIs with alternative values for a parameter	114	51.3%
APIs with default values for parameters	99	44.6%
APIs that state the datatype of the parameters	61	27.5%
APIs with coded values for a parameter	55	24.8%
APIs with boolean parameters	39	17.6%

TABLE 6.3: Survey 1 - Input Parameters

The fact that a lot of APIs use alternative values for one parameter (for example, a range of 1, 2 or 3) and coded values (for example, 'en' for English) makes the API invocation even more challenging. For the automated invocation of single APIs, the input data has to be transformed in the correct format, which can be very difficult, since sometimes the lists with alternative or coded values are not provided. For the invocation of mashups, the transformation between the outputs of one API and the inputs of the next one has to be defined. Currently, this work requires extensive manual effort and the adaption of existing Web service invocation approaches is hindered by the under-specification and the variability of the parameters. Therefore, client application developers are required to test-invoke APIs, in order to determine the missing information in the documentation, and to implement custom solutions that handle the different processing options.

This situation is aggravated by the fact that two thirds of the APIs do not even state the datatype of the input parameters. As a result developers need to determine the proper input format by making assumptions or through trial-and-error. In addition, the development of new client applications is made difficult, since the datatype information is simply not available. The lack of

datatype specification hinders the automated processing, imposing the need to manually interpret the documentation, instead of having a description that already supports the machine-to-machine communication, such as WSDL.

6.3.5 Output Formats

As can be seen in Table 6.4, there are two main common output formats – XML and JSON. XML is provided in 85% of the cases and JSON in 42%, while more than one third of the APIs provide both. Further output formats include HTML, CVS, RDF, Text, object, RSS, GFF, Serialised PHP, Tab, YAML. These results show that providing support for the use of XML and JSON addresses the vast majority of the APIs.

It is also important to point out that only about 6% of the APIs deliver the output in RDF, enabling the communication based on semantic information. This plays an significant role for the scope of this thesis, since one key step along the way of enabling Open Services on the Web, based on the integration of services on the Web and Linked Data, is the support for exchanging and communicating via a format that can capture semantics (i.e. RDF). As can be seen, currently the adoption of RDF for APIs is still very limited.

Description	Number	In %
XML	80	36%
XML and JSON	53	23.9%
XML and other	34	15.3%
XML, JSON and other	23	10.4%
only JSON	12	5.4%
only other	14	6.3%
JSON and other (except XML)	6	2.7%
RDF	13	5.8%
Total XML	190	85.6%
Total JSON	94	42.4%

TABLE 6.4: Survey 1 - Output Formats

Even though, HTTP provides a standardised way of determining the required output format (via content negotiation [FGM+99]), as part of the HTTP request, most providers disregard this and the majority of the APIs specify how the results should be structured in two main ways. Either the API provides a separate operation for every output format or it is determined through a parameter (for example, `http://my.example.org/.../getXmlNews?user=aUser` and `http://my.example.org/.../getNews?user=aUser&format=xml`).

6.3.6 Invocation Details

In this section we describe our findings in relation to the invocation details commonly provided in API descriptions. The collected data is of crucial importance, since it has a direct impact on the usability of the APIs.

Description	Number	In %
Provide HTTP method	134	60.4%
Provide invocation URI	214	96.4%

TABLE 6.5: Survey 1 - Invocation Details

Table 6.5 shows that almost all descriptions provide the URI for invoking the API, while the remaining 3% do not include it as part of the documentation. In such cases developers have to assume that the invocation URI is the same as the documentation one or seek further support. Either way, this requires additional manual effort. Not providing the invocation URI, would be unthinkable if a formal interface description language were used, such as WSDL for Web services. This is also true for not stating the HTTP method, which holds for two thirds of the APIs. One possibly reason is that providers assume that the method to use is GET, especially for APIs that can be invoked directly through parameterising the URI. If the description of APIs were guided by standards, this form of underspecification would not be allowed. This kind of crucial but commonly omitted Web API characteristics highlights the need for the work carried out in this thesis (see Chapter 7).

6.3.7 Authentication Details

Authentication Mechanisms	Number	In %
API Key	89	38%
HTTP Basic	32	14%
Username and Password	19	8%
OAuth	14	6%
Web API Operation	12	5%
HTTP Digest	11	5%
API Key in Combination with Other Credentials	5	2%
Session Based	5	2%
Other	2	1%
Authentication Only for Data Modification	4	2%
Offer Alternative Authentication Mechanisms	16	7%
No Authentication	46	19%

TABLE 6.6: Survey 1 - Common Web API Authentication Approaches

Our analysis also shows that more than 80% of the APIs require some form of authentication[20] (Table 6.6). As can be seen, using an API key (also called "developer key", "developer token", "token Id", "user Id", "user key") is by far the most common way of authentication[21] (38%). It is followed by 19% of APIs, which do not require any authentication. HTTP Basic and HTTP Digest [FHBH99] are not used as often (14%, 5%), while about 6% of the APIs use OAuth [MA10] and 5% implement their own operations, which need to be called, before being able to invoke other operations. There are some APIs, which require authentication only for operations, which perform data modification but require no authentication for only reading resources.

In summary, 3 out of 4 APIs require some form of authentication, which means that developers would have to sign up with providers for acquiring the appropriate credentials. In addition, there is no established approach for Web API authentication but rather a landscape of different approaches. Also, about only a quarter of the APIs use a mechanism that protects the user credentials and does not transmit them directly in plain text (HTTP Basic, OAuth and HTTP Digest) . This shows that providers are not so much concerned with securing the user credentials and do not invest implementation work in securing the message transfer but rather prefer to focus on controlling resources usage. This is verified by the fact that only about 10% of the Web APIs use signatures and encryption (OAuth and HTTP Digest).

Transmission Medium	Number	In %
URI	117	70%
HTTP Header	45	27%
URI or HTTP Header, Depending on the Type of Authentication and HTTP Method	6	3%

TABLE 6.7: Survey 1 - Way of Transmitting Credentials

Table 6.7 shows the most commonly used ways for transmitting authentication credentials. As can be seen, 70% of the Web APIs send authentication information directly in the URI, while less than one third require that a special HTTP header is constructed. This means that even if Web APIs require authentication, most of them do not need a custom client but can rather be invoked directly from a Web browser. These numbers are similar for invocation in general, where about one third of the APIs require the construction of the HTTP request, while the rest can be invoked by using the URI.

[20] Sum over all authentication types, adding up to 81%.

[21] See Section 4.2 for a detailed description of all authentication approaches.

6.3.8 Additional Documentation

Finally, this section describes API description features, which are not strictly necessary for directly supporting service tasks such as discovery or invocation, but are useful when implementing and using APIs. As Table 6.8 shows, more than 75% of the APIs provide example requests and responses. These give valuable information about the structure and the form of the request as well as of the retrieved results and, therefore, ease development work.

Description	Number	In %
APIs that provide an example Request	186	83.8%
APIs that provide an example Response	167	75.2%
APIs that describe the Error messages	118	53.1%

TABLE 6.8: Survey 1 - Complementary Documentation

We also found out that about only half of the APIs describe the used error codes. This represents a problem for the implementation of client applications, since developers cannot determine what went wrong and whether the error is due to an incorrect invocation, to missing credentials, etc.

6.3.9 Summary of Results

This section provides a summary of the findings of the first survey and derives a number of important conclusions, characterising the Web API landscape. The made observations are an important step towards gaining a clear picture of the development process, used technologies, available information, richness of the descriptions, etc., in the context of Web APIs. These serve as a foundation for developing approaches and software solutions that lead to a more automated API use.

1) Web API directories, like ProgrammableWeb, which are based on manual input contain inaccurate or outdated details.

This result points out one of the main challenges faced by current Web API repositories. Since the API descriptions are published and updated manually by users, some of the entries are not up-to-date or no longer exist. Therefore, there is a need for developing solutions for a more automated way of collecting, publishing and updating API descriptions.

2) Few APIs are highly reused, whereas most APIs, are used in very few or no mashups at all. In addition, there is no correlation between the level of reuse of APIs and their granularity.

The level of reuse, as indicated by the number of mashups per API, is a very important characteristic of the current Web API landscape. Since we have no direct information about how many of the existing APIs are actually being used, the number of mashups is an indirect indication for that.

3) There are three main types of Web API descriptions (RESTful, RPC-style and hybrid). Developers prefer to describe APIs in terms of operations, rather than resources.

This means that each type of Web API requires separate invocation solutions, otherwise the support provided by any approach would have only limited coverage. Currently, with the exception of some frameworks, with limited functionality (see Section 4.1.3), API invocation is based on custom solutions, which have a low level of reusability and do not contribute to the automation of a shared API invocation process.

The fact that most developers prefer to describe APIs in terms of operations, disregarding REST principles may be explained by looking at popular ways for defining interfaces and frequently used programming languages in general, which are commonly based on operations and methods. Therefore, developers with previous knowledge of interface description languages and a background in programming intuitively tend to formulate Web APIs in terms of operations, rather than resources that are manipulated through the HTTP methods. This, however, still needs to be confirmed by directly questioning Web API developers.

4) The current proliferation of Web APIs cannot be attributed to the use of RESTful APIs.

The popularity and use of Web APIs, being offered by websites, is not driven by a particular REST features, since RESTful APIs account for only about 30%.

5) API reuse in not driven by the particular type of Web API description (RESTful, RPC-style or hybrid).

We base this conclusion on the fact that the mashups percentage distribution matches almost exactly the Web API description type distribution. Our data shows no indication of RESTful APIs having a leading role in determining whether APIs are used in mashups or not.

6) The description of input parameters is very diverse, allowing for the use of default values, coded values, alternative values and optional parameters.

Service tasks that predominantly rely on the input information, such as discovery, composition and invocation, gain complexity, since the presence of some parameters is non-restrictive and

the input data has to be transformed into coded or alternative values. Therefore, the approaches, which aim to support the use of Web APIs, should be able do deal with the diversity of the input parameters. This is especially true for invocation, which would require the development of an integrated view on all theses diverse input forms.

7) XML and JSON are establishing themselves as the main output formats.

Even though there are no guidelines for the format of the output, currently most APIs give their results either in XML or JSON. Therefore, providing support for using and processing only these two formats, would already enable the handling of the majority of the APIs' output.

8) Authentication plays a major role in the context of supporting automated Web API use.

More than 80% of the APIs require some form of authentication. Therefore, authentication is a vital part of the invocation process and any approach for supporting the use of APIs and mashups that disregards authentication, has very limited applicability. Currently, developers have to sign up with multiple providers in order to acquire credentials necessary for APIs participating in mashups or restrict the implementations to APIs, which are based on shared credentials such as OAuth [MA10].

9) API documentation is frequently characterised by under-specification.

Our data shows that two thirds of the APIs do not state the datatype of the input and 40% of the APIs do not state the HTTP method. If a standard interface description language, such as WSDL, were used to describe Web APIs, not specifying these details would be unthinkable. However, since there is no common IDL, under-specification is very common.

Looking at the different results provided in this section, it becomes obvious that currently the Web API landscape is very heterogeneous and there is no such thing as a 'standard Web API documentation' or standard practices for that matter. Without a doubt, all descriptions contain common pieces of information, which are required for the support of main service tasks, such as discovery, composition and invocation. However, since Web API development is not guided by standards, the diversity spreads from the structure and the form of the documentation up to the technological principles used behind the implementation. Therefore, currently the use of APIs requires extensive manual effort and the development of automated approaches is very challenging. With our study we provide the foundation for the development of approaches that better support API use by contributing to a clearer picture of the current Web API landscape. Furthermore, the gathered insights can be used to guide the development of applications and solution approaches, or even serve as the basis for launching activities that would encourage providers to offer more REST-conforming APIs.

6.3.10 Discussion

In this section we reflect on a number of further trends and correlations that we discovered while conducting our Web API analysis. In particular, we describe how APIs from the same domain tend to have some similar features.

One interesting correlation that we detected is that APIs from the same ProgrammableWeb category tend to have the same type of description. For example, all bookmarking APIs were RPC-style, while all project management ones were RESTful. This is also true for most of the categories, where we found out that the majority of the APIs have the same type[22]. This might be due to developers investigating competing providers and their services and, therefore, being influenced by the way APIs with similar functionalities are structured and described.

In addition to having similar types of descriptions, we discovered that APIs from the same category usually have similar authentication mechanisms. For example, most APIs from the government or health information domain require no authentication, while APIs for job search and general search commonly use an API key. This can be attributed to the tendency that certain domains should naturally be very accessible, while others related to more private or confidential information, should be supported by stronger authentication measures.

The survey also provided some important information about the Web API description forms. In particular, none of the analysed APIs used WSDL [W3C07a] or WADL [Had06] and the majority of the APIs are documented directly in HTML webpages. In addition, some of the descriptions were in PDF, which requires downloading the documentation and makes crawling for APIs and automated processing more difficult.

6.4 Second Web API Survey

The second Web API survey was conducted in February 2012, almost two years after the initial study. It was carried out to cover the exact same APIs that were investigated in the first study. Our main goal was to revisit the results of the first study and to identify any changes or trends in the analysed characteristics. In particular, we covered the 222 APIs from the first study and added the top 10 most popular APIs in order to compensate, in the case that some entries no longer existed. In fact, we found out that 4 of the APIs were no longer available in the directory, resulting to a total of 228 to analyse.

Currently ProgrammableWeb is still the most popular and commonly used API directory, which at the time of the second study provided information about 4796 Web APIs and 6404 mashups (numbers from mid-January 2012). The structuring of the directory has not been significantly

[22] All the gathered data is available at `http://purl.oclc.org/NET/WebApiSurvey/`.

altered, still including the same options for keyword search and browsing by category, data format, company (provider) and protocols/styles. With the growing number of the stored APIs, the number of categories used to the describe them has increased from 51 to 55.

6.4.1 Methodology

In contrast to the first study, which had the purpose to provide a general overview of the state of the Web APIs and their general characteristics, this survey aims to review and further explore the initially conducted analysis and also to capture the developments in the ways of describing and providing APIs. In addition, its goal is to investigate some correlations and interesting features that were identified while completing the first study. Therefore, all the previously used criteria were included and some new ones were added. Based on this set up, we are able to compare the results of the two studies and discuss trends and developments that have taken place over the past two years.

In particular, the analysis included all the APIs from the first study, in order to be able to reflect on how their characteristics have changed, and was extended with an additional 10 entries from the list of most popular APIs. We chose to include the most popular APIs, instead of adding randomly chosen APIs from each category, because in this way the results will reflect the characteristics of the most commonly used APIs. In total, the second Web API survey covers 7% of the APIs listed in ProgrammableWeb, which are marked as using REST (3309 APIs at the time of the study). Similarly, to the first study, it was done manually but this time also assessed by a supporting system (see Section B.2).

Each Web API documentation was manually analysed in terms of the same six main groups of features, whose individual properties were extended and refined. These include general Web API information, type of Web API, input details, output details, invocation details and additional documentation. The Web API analysis was conducted manually. However, this time it was done with the help of an especially designed and implemented Web application, described in more detail in Section B.2. Each Web API was examined in terms of:

1. **General Web API information** – we recorded all the characteristics from the first study, including the name of the API, its description, the date when the documentation in ProgrammableWeb was updated, the assigned categories and tags, number of mashups, URL and number of operations. This time we also checked whether the documentation URL actually points to the provider's documentation webpage (correctness of the documentation URL).

2. **Type of Web API** – we checked whether the API description is RESTful, RPC-style or hybrid. During the second study we also recorded if the invocation URL reflects the hierarchical structure of the resources.

3. **Input details** – in addition to the characteristics analysed in the first study (default parameters, optional parameters, coded parameters, alternative values for parameters, boolean parameters, stating of the datatype), we also collected details on whether the input is a complex object and whether there are any links between the outputs and inputs of different operations.

4. **Output details** – we recorded the form of the output (for example, XML or JSON) and whether it is determined via an input parameter. This time we also checked if there is an output schema definition provided.

5. **Invocation details** – in addition to the collected invocation details during the first survey (provisioning of the HTTP method and the invocation URI, type of authentication and way of transmitting the input details and the authentication information), during the second survey we also checked whether the invocation URI uses templates, parameters and version number. Furthermore, we also recorded for which operations authentication is required.

6. **Additional documentation** – we examined the provisioning of example request, example response and a list of error messages/codes. For this study, we also checked if the used errors are the standard HTTP errors or are custom ones used.

The second Web API survey includes the same groups of features as used in the initial one, with the addition of a few new features that had been identified as relevant. For example, during the first survey it became evident that sometimes the URL listed in ProgrammableWeb, which is supposed to point to the documentation of the API, actually points to the provider's website or to a different website instead. In addition, some of the APIs from the directory were no longer available. We decided to record this information in order to be able to make statements about the accuracy of the information provided in the directory but also about the dynamic nature of the entries. Naturally, the main focus was on gathering details about different API aspects that are relevant for developing a method that supports a more automated API use. While the first study provides an initial overview of the heterogeneous API landscape, the second analysis was more targeted towards gathering input for refining and improving the description models, which are detailed in Chapter 7. This includes a more detailed analysis of the different Web API types, gathering of data about the input message, output provisioning and presentation, and error handling. We align with the features that are present in interface description languages (IDLs), as they are considered essential for enabling invocation [Pau09]. In addition, we consider the

complementary documentation, which provides details on how the communication between the client and the server is realised, and what are the possible errors that can occur.

The analysis approach of the second study was conducted in a slightly different manner as opposed to the initial one. A collection of all Web API entries that were chosen for the survey, including all the available details from ProgrammableWeb, was stored in a triplestore datastore (we used Sesame[23]). The APIs to analyse included all the entries from the first study. We did not chose the APIs randomly because we wanted to be able to compare the results for the exact same set of entries, as originally used. In this way we are able to revisit the data collected from the first survey, confirm the made statements and explore further relevant features.

The second API survey was based on the following steps – each API entry is opened in an especially designed survey Web application (presented in detail in the Section B.2) and the details available from ProgrammableWeb are directly presented for validation. The person completing the survey has to fill out four Web forms, based on free-text, drop-down selections, radio buttons and checkboxes, and finally click on a button to submit the results. The new survey system can be configured to include only a sub-set of all the analysis features and makes it possible to crowdsource the evaluation process to people from distributed locations. Each survey entry is user-specific (based on recording their email address), enabling the collection of data inputted by a single user or a group of people. The results presented here are based on the full set of features and were completed by a domain expert.

The main objective of the second survey is to provide insights on improving the support for automating the use of Web APIs, therefore, more attention was paid to invocation-relevant characteristics and less space was devoted to collecting general API information. The results of the second Web API study are presented in the following section.

6.4.2 General Web API Information

The analysed general Web API information includes validation and recording of some of the details provided by ProgrammableWeb about each API. In addition to the characteristics analysed during the first study, we also determine the correctness of the documentation URL, let the person completing the survey freely assign categories to the API, record a set of tags and count the number of operations.

As we discovered, in some cases the survey cannot be completed because the API is no longer available or because the documentation cannot be found. We record these cases and the corresponding reason for not being able to fill-out the question forms. These results are of special

[23] `http://www.aduna-software.com/technology/sesame`

interest for the subset of APIs included in the first study because we can determine what percentage of the APIs were taken offline during the two years between the surveys. Table 6.9 provides the numbers for some of these features, including a comparison with the results from the first survey (columns marked with 'S1').

Description	Maximum	S1	Minimum	S1	Average	S1
APIs per Category	53	12	1	1	8	4
Number of Operations	100 < # < 200	# > 200	1	1	11 < # < 50	15.5
Number of Mashups	602	506	0	0	8	6.4

TABLE 6.9: Survey 2 - General Web API Information

As part of the analysis, we determined that the distribution of the categories was quite different in comparison to the first study (maximum of 53 entries for one category and minimum of 1 for a total of 53 used categories), while the most popular categories were *social*, *tools* and *other*[24]. This is due to the fact that the overall number of APIs in ProgrammableWeb has increased, which results in an increase of the APIs assigned to each category.

In the initial study, if we could not complete the survey for a given API, we simply moved on to the next one. This time we recorded the number of cases, in which the URL listed in the directory did not point to the actual Web API documents. This was the case with 97 APIs, which accounts for a surprising 42% of the analysed APIs. This means that developers cannot rely on the directory to find the link to the provider's documentation and in half of the cases have to search for it themselves. In addition, the survey could not be completed for a total of 55 APIs[25]. There are two main reasons for not being able to complete the survey – first, the APIs is no longer available (36 APIs), second, the documentation cannot be found or is not available (18 APIs). What is more interesting is that out of the APIs used to complete the first survey almost 25% were no longer available and 4 APIs were removed from the directory. This demonstrates that the API landscape is very dynamic, with a large portion of APIs being taken offline and new ones being published.

Similarly to the first survey, we take the number of mashups as a measure for the reuse of Web APIs and use it as an indicator for factors that influence the reuse of APIs. Since ten of the APIs are taken from the list of the most popular APIs in ProgrammableWeb, we base this analysis only on the remaining part of the studied APIs, i.e. we consider only the APIs from the first survey. Otherwise the average number of mashups per API would rise from 8 to 19.

The second study confirms again that a few APIs are highly reused, whereas most APIs are used in very few or no mashups at all. In particular, there are 101 APIs with 0 mashups, 85 APIs with

[24]The numbers for 'APIs per Category' are based on the complete ProgrammableWeb directory and not on our test API subset.

[25]This number should not be confused with the 4 APIs, which were excluded from the second survey, because they are no longer listed in ProgrammableWeb.

1 to 4 mashups and 32 APIs with 5 to 602 mashups. The API with most mashups is Flickr[26] (overall in ProgrammableWeb is Twitter with 690 mashups). Overall, the average number of mashups per APIs has increased, from 6 to 8. This shows an overall tendency of building more mashups, since the total number of mashups over all analysed APIs this time was 1879 (1350 for the first study).

As part of collecting general information about Web APIs, we recorded the number of operations for each API. It is important to point out that we used the results of the first study to adjust the features used in the second one. For example, we used a range, instead of an exact number, for recording the number of operations (1, 2-10, 11-50, 51-100, 101-200, 200+). The limits of the ranges are based on the results of the initial survey, which indicated a distribution of the numbers matching these ranges. This adjustment helped speed up the analysis of each API, since counting the individual operations was very time consuming during the first study. The results show that the majority of the APIs have between 2-10 (37%) and 11-50 (35%) operations, while only few APIs have only one operation (17%) or more that 50 (11%) operations. This leads us to the conclusion that most APIs have fewer operations.

The results of the second Web API survey also confirm the overall problem that since all details are added manually to the Web API directory, some of the feature descriptions were not always accurate. This is especially true for the URL of the documentation, which had sometimes been moved or was no longer available, and for the authentication information, which was very often inaccurate. This yet again confirms the difficulties resulting from using directories based on user entries and highlights the need to develop approaches for automatically collecting and extracting API descriptions from the Web.

6.4.3 Type of Web APIs

We used the same three types of Web APIs, which were defined and identified during the first Web API survey – RESTful, RPC-style and Hybrid. It is important to note that the APIs classified as RESTful in the survey are the ones that closely follow the representational state transfer (REST) paradigm [Fie00] and represent a subset of the APIs labelled as "REST" in ProgrammableWeb. This occurs because the APIs marked as "REST" in ProgrammableWeb actually include RPC-style and hybrid ones as well. The term *RESTful services* is often misused to denote Web APIs in general, instead of only the ones conforming to the REST principles.

Table 6.10 summarises the overall distribution of the types of APIs. As can be seen, similarly to the results of the first study, currently almost half of the Web APIs are RPC-style and about one third are RESTful. The hybrid APIs represent only about 16% of the analysed data. This percentages demonstrate that REST principles still remain to be widely adopted by API providers.

[26] `http://www.flickr.com/services/api/`

Description	In %	S1 In %
RPC-Style	47.1	47.8 %
RESTful	36.6	32.4 %
Hybrid	15.7	19.8 %

TABLE 6.10: Survey 2 - Type of Web APIs

This distribution is very similar to the one of the first study, which indicates that providers are not eager to change the existing implementation in order to adopt a REST view on APIs.

In comparison to the first survey, we aimed to explore the characteristics of RESTful APIs in more depth and included the tracking of features such as the hierarchical structuring of the resources, statelessness of the request, and determining the output format via content negotiation[27] [FGM+99]. It needs to be pointed out that, similarly to the first survey, we were only able to explore the Web APIs based on the interface properties captured in the documentation. Since the REST principles apply to the actual architectural style, what we see in the documentation is an indirect reflection of the underlying implementation. In particular, we analysed the following characteristics:

- RESTful APIs with hierarchical structuring of the resources – REST conform;
- RESTful APIs with scope definition (URI with parameters) – REST conform;
- RESTful APIs with parameter-determined output format (violating content negotiation), as opposed to RESTful APIs with content negotiation-based output format – not REST conform;
- RESTful APIs using client-specific information, such as sessions – not REST conform.

The use and the stating of the HTTP method, which is also relevant in the context of RESTful APIs, is discussed in Section 6.4.6 as part of the invocation details.

Description	In %
RESTful APIs with Hierarchical Structure of Resources	80.9%
RESTful APIs with scope definition	61.9%
RESTful APIs with output format via content negotiation	19%
RESTful APIs using client-specific information	0%

TABLE 6.11: Survey 2 - RESTful Web APIs

Considering the analysis of REST-specific characteristics (see Table 6.11), we found out that the majority of the RESTful APIs use URIs that reflect the hierarchical structuring of the resources

[27] REST principles are discussed in detail in Section 3.2.

(81%). In addition only about two thirds of the APIs use parameter query values in order to restrict the scope of the retrieved results. These two characteristics conform to REST principles and indicate that when providers chose to define interfaces in terms of resources instead of operations, they follow the guidelines for defining the corresponding URIs [RR07].

In contrast, instead of using the HTTP header properties in order to specify the expected output format (content negotiation), the majority of the APIs (more than 80%) do this directly as part of the URI, for example through parameters or file extensions ('.xml'). In the context of developing client applications, this means that instead of relying on HTTP and handling the requested output format directly, the developed solutions need to handle the different formats based on the passed parameter values. This leads to additional overhead and the development of individual custom implementations, which could be avoided by relying on the HTTP standard. This example is indicative for the trend that even if providers are trying to develop RESTful services, the level of conformity to all principles is relatively low and they stick to simple characteristics such as the hierarchical structuring of the access points.

6.4.4 Input Details

In contrast to the first survey, where only the diverse characteristics of the input parameters were analysed, this time we also tracked the way of sending the input, for example, as part of the URL or in the HTTP body, and also recorded the cases where the input is not parameter-based but is rather given in a more complex format, such as JSON or XML. These new insights provide guidance on crafting an API description model that captures the input characteristics and lead to requirements related to the way the actual HTTP request needs to be formed, as part of completing the API invocation process.

Description	Number	In %	S1	S1 In %
APIs with optional parameters	95	55.2%	136	61.3%
APIs with required parameters	95	55.2%	na	na
APIs with alternative values for a parameter	110	63.9%	114	51.3%
APIs with default values for parameters	89	51.7%	99	44.6%
APIs that state the datatype of the parameters	49	27.9%	61	27.5%
APIs with coded values for a parameter	60	34.9%	55	24.8%
APIs with boolean parameters	58	33.7%	39	17.6%

TABLE 6.12: Survey 2 - Input Details

Table 6.12 summarises the results regarding input details. As can be seen, about 55% of APIs use optional parameters, while 55% use required parameters. The percentages of APIs using optional and required parameters do not add up to a 100 because a parameter was recorded as optional or required only if this was explicitly stated in the descriptions. For example, if an

API has five parameters, two of which were marked as required and the remaining three were not described as optional, the API was counted as using only required parameters. We make this differentiation for a number of reasons but mainly because we wanted to collect results that directly reflect the inconsistency or incompleteness of the documentation. First, if we use natural language processing techniques to analyse the documentation and automatically extract API properties, we would not be able to recognise if the property is optional or required, since this is simply not stated. Second, in order to check whether a parameter is truly optional and what the corresponding default values and effects on the retrieved results are, we would need to perform test invocations. This again highlights the need for guidelines and standards on describing APIs and demonstrates how their lack results in underspecification and misleading information.

Similarly to the first study, more than two thirds (72%) of the APIs do not state the datatype of the input parameters. Therefore, developers need to determine the proper input format by making assumptions or through trial-and-error. This is crucial not only for providing correct input values, but more importantly for using existing datasets as input sources, which would need to be transformed in the correct format before they can be used. Similarly, this hinders the integration with further APIs and as part of existing applications. Overall, the percentage values for the different input parameter characteristics are very similar to those of the first study, therefore, we cannot conclude that there is an improvement in the level of underspecification or that certain formats are preferred over others. Therefore, providers are not necessarily eager to up update or change the offered documentation and to enrich it with missing details. One would expect that user complaints and feedback would encourage completeness of the documentation, therefore, resulting in a change in the percentage values. However, such trend is not to be observed.

Transmission Medium	Number	In %
URI	135	78.5%
HTTP Header	1	0.6%
HTTP Body	32	18.6%
Mixed	3	1.7%

TABLE 6.13: Survey 2 - Way of Transmitting Input Parameters

Concerning the way of transmitting the input parameter values (see Table 6.13), currently with the majority of the APIs, this is done directly as part of the invocation URI (almost 80%). A small number of APIs require sending input data directly in the HTTP Body, while only a few use the Header. These values are indicative in the context of developing solutions that support the automation of the invocation task, since they show that providing a mechanism that relies on defining the URI and the corresponding input values, would already provide support for the majority of the APIs.

6.4.5 Output Details

Confirming the results of the first study, XML and JSON are being established as the two most commonly used output formats (see Table 6.14). This is especially true for XML, which is sometimes used without even explicitly stating in the API documentation that the output is XML-based. The gathered data show that providing support for the use of XML and JSON addresses the vast majority of the APIs. Again these results are very similar to the results of the first survey, with a slight growth in the overall use of JSON (growth of 10%).

Description	Number	In %	S1	S1 In %
XML	65	37.8%	80	36%
XML and JSON	46	26.7%	53	23.9%
XML and other	8	4.6%	34	15.3%
XML, JSON and other	27	15.7%	23	10.4%
only JSON	10	5.8%	12	5.4%
only other	12	6.9%	14	6.3%
JSON and other (except XML)	5	2.9%	6	2.7%
RDF	11	6.4%	13	5.8%
Total XML	144	83.7%	190	85.6%
Total JSON	91	52.9%	94	42.4%

TABLE 6.14: Survey 2 - Way of Transmitting Input Parameters

Since XML is by far the most commonly used output format, we recorded how often there is a schema definition provided as part of the description of the output. The results show that frequently there is an example but the cases where the structure of the output is formally specified are rather an exception than a rule. In particular, only about 11% of the APIs provide a schema definition for the output, or 14% of all APIs that use XML. This numbers are still very low, which means that we cannot rely on the information in the documentation in order to be able to process directly the output and some additional interpretation would be necessary.

Description	Number	In %
Via input parameter	36	20.1%
As a file extension	34	19.8%
As part of the URI	15	8.7%
Via content negotiation	18	10.5%
Unclear/not specified	119	40.9%

TABLE 6.15: Survey 2 - Way of Requesting the Output Format

During the first study, we discovered that there are a number of different ways of requesting a specific output format, which did not align with the standardised way of determining it – via content negotiation as part of the HTTP request. As can be seen in Table 6.15, the most commonly used solution is via a parameter with the particular format as value (`www.example.`

`com/api/getNews?format=xml`), by adding the format as a file extension at the end of the URL (`www.example.com/api/getNews.xml`) or by providing a separate operation for every output format (`www.example.com/api/getXMLNews`). Especially for RESTful services, it would be expected that the format of the output is determined via content negotiation by using the HTTP header. However, the results clearly show that API providers prefer a more visible way that requires less knowledge of the underlying technology stack. The percentage values do not add up to 100, since the remaining portion of the APIs did not have multiple output formats or did not explicitly specify how the output format was determined.

6.4.6 Invocation Details

Table 6.16 shows that in the majority of the cases the documentation provides the URI for invoking the API, while only about 60% state the HTTP method to be used. These results are very similar to the ones obtained two years ago.

Description	Number	In %	S1	S1 In %
Provide HTTP method	102	59.3%	134	60.4%
Provide invocation URI	154	89.5%	214	96.4%
Invocation URI composed through templates	121	70.3%	na	na
Invocation URI uses query parameters	133	77.3%	na	na
Invocation URI includes version number	40	23.2%	na	na

TABLE 6.16: Survey 2 - Invocation Details

In addition to the features analysed in the first study, this time we also collected some details on how the invocation URI is constructed. In particular, we found out that 70% of the URIs are based on using templates, where a placeholder in the path is substituted by a particular value in order to retrieve the wanted output (`www.example.com/api/{date}/news`). Furthermore, the majority of the APIs use parameters (77%) (`www.example.com/api/getXMLNews?topic=tv`), therefore providing support for parameterised URI as part of the Web API description model is crucial. This affects both the way that the actual invocation endpoint is specified and influences the input provisioning, which needs to include a mapping between the actual values and the parameters in the URI. We also found out that 23% of the APIs use the version number as part of the invocation URI, in order to differentiate between previous and current versions. In this way multiple versions of the same API can coexist and the implementations realised with older versions can continue to be used and do not necessarily have to be recoded every time the API is updated. This is an interesting trend but it is yet to be seen if it finds wider adoption.

6.4.7 Authentication Details

Authentication plays a crucial part in enabling the usability of APIs, since without providing support for it, the actual API invocation cannot be completed. This is why we devote special attention to the commonly used authentication approaches and the ways of transmitting the required credentials as part of the HTTP message. Our analysis also shows that 80% of the APIs require some form of authentication (Table 6.17). These results are very similar to the data collected during the first study and confirm the continuous importance of authentication in the context of using Web APIs.

Authentication Mechanisms	Number	In %	S1	S1 In %
API Key	53	30.8%	89	38%
HTTP Basic	37	21.5%	32	14%
Username and Password	13	7.5%	19	8%
OAuth	22	12.8%	14	6%
Web API Operation	8	4.6%	12	5%
HTTP Digest	10	5.8%	11	5%
API Key in Combination with Other Credentials	20	11.6%	5	2%
Session Based	3	1.7%	5	2%
Other	1	0.6%	2	1%
Authentication for All Operations	147	85.5%	na	na
Auth. Only for Data Modification or Some Operations	8	4.6%	4	2%
Offer Alternative Authentication Mechanisms	35	20.3%	16	7%
No Authentication	35	20.3%	46	19%

TABLE 6.17: Survey 2 - Common Web API Authentication Approaches

As can be seen, using an API key continues to be by far the most common way of authentication (31%). It is followed by 21% of APIs, which use HTTP Basic, showing an increase in the numbers in comparison to the first study. As in the first study, about 20% of the APIs require no authentication.

The results confirm the trend that there is not one established authentication approach, but the majority of the APIs require some form of authentication. There is no significant development towards adopting one particular authentication approach, even though, OAuth shows a doubling of the percentage values, and HTTP Basic shows an increase as well.

Each authentication approach can be realised with a different set of credentials that are sent via different parts of the HTTP request. In order to be able to provide support for describing the authentication approach used by an API, we also studied the commonly used ways for transmitting authentication credentials.

As can be seen in Table 6.18, about 70% of the Web APIs send authentication information directly in the URI, while 23% require that the HTTP header is constructed. Therefore, for the

Transmission Medium	Number	In %	S1	S1 In %
URI	95	69.3%	117	70%
HTTP Header	32	23.4%	45	27%
HTTP Body	8	5.8%	na	na

TABLE 6.18: Survey 2 - Way of Transmitting Credentials

majority of the APIs it is sufficient if the invocation URI is constructed properly, while only about one third of the APIs require the construction of the HTTP request, inserting input and authentication credentials in the body or the header. These numbers are very similar to the results of the first API survey.

6.4.8 Additional Documentation

As can be seen in Table 6.19, the majority of the API documentations include example requests and responses. These details can be used to complete missing information that is not given as part of the input, output or endpoint documentation. In addition, they can also serve as a basis for deducing the structure of the request and the format of the results. The percentage values for the two surveys are very similar, therefore, we cannot say that there is a trend towards providing a more detailed documentation, that includes further information such as examples.

Description	Number	In %	S1	S1 In %
APIs that provide an example Request	153	88.9%	186	83.8%
APIs that provide an example Response	137	79.6%	167	75.2%
APIs that provide description of Errors	94	54.6%	118	53.1%
APIs that use Custom Errors	50	29.1%	na	na
APIs that use standard HTTP Errors	44	25.6%	na	na

TABLE 6.19: Survey 2 - Complementary Documentation

In the first study we did not investigate in much detail the description of errors and the use of standard HTTP errors. In this survey we were able to determine that about half of the APIs provide descriptions of the errors that can occur when calling the API. Overall, only about 25% use standard HTTP errors.

Unfortunately a large portion of the APIs (30%) use customs errors, where a 200 OK HTTP code is returned to the client but instead of the expected data, the output contains an error description. This makes the realisation of client applications more difficult, since developers would need to handle custom exception implementations, in order to be able to determine what went wrong and whether the error is due to an incorrect invocation, to missing credentials, etc. Furthermore, this presents a challenge for developing approaches that are capable of handling errors as part of

the API invocation process, since the error cannot be recognised directly based on the returned status code but instead, the actual output needs to be processed.

6.4.9 Summary of Results

Based on the analysis of the data collected as part of the second API study, we are able to make some conclusions about the general state of APIs on the Web. While the main focus of the first survey was on gaining an initial clear picture of the used technologies, available information, richness of the descriptions, etc., the goal of the second study was to observe changes and developments in the results of the first study and also to investigate in more detail the features that were identified as influential to the use of APIs. In addition, based on comparing the two datasets, we were able to identify some trends in the ways of providing API documentation. Overall, the collected results are extremely valuable as an input for designing a Web API description model that is able to capture the diversity of formats and structure of the existing APIs in a unified way, thus providing the basis for developing approaches for a more automated Web API use.

The results of the second study, to a large part, confirm our findings from the first one, providing some further insights. Web API search still has to be done manually and existing directories continue to have inaccurate and outdated entries [LHPD12] (Section 6.4 point (1)). Furthermore, we were able to confirm that the entries are not always up-to-date and, more importantly, that some APIs no longer exist but are still kept in the directory.

1) The Web API landscape is very dynamic, with many new APIs being offered but also a number of APIs no longer being available.

Given the statistics available about the growth of numbers of APIs and mashups in the ProgrammableWeb directory, it is easy to assume that every day there are more and more providers that choose to offer access to existing resources of functionalities through programmable interfaces. Indeed, comparing the entries in the directory now[28] with those two years ago, the number of APIs has almost doubled. However, by analysing the same set of APIs used for the first study we discovered that 24% of them are no longer available. This gives a new perspective over the Web API landscape, which is very dynamic with new APIs being offered but also a lot of the already existing ones being taken offline.

The new data also confirmed the tendency of using a few popular APIs in a multitude of mashups, as opposed to most of the APIs being rarely used as part of mashups or not at all. Therefore, in order to be able to reflect on the types of documentation that developers are most

[28] At the time of the writing, applies to the state of the directory in early 2012.

frequently faced with and to be able to provide adequate support as part of an API description model, we need to focus on analysing the features of the most popular APIs.

There continue to be three types of Web APIs (RESTful, RPC-style and hybrid) (Section 6.4 point (3)) and the majority of the APIs are still based on operations and not resources. In addition, despite the growing awareness of the REST principles and the effort of some providers to expose truly RESTful APIs, the analysis shows that there are still some characteristics such as the definition of the expected output format via a parameter or the use of custom errors that break with protocol standards and REST guidelines. This raises the questions whether REST is not counterintuitive for developers with previous knowledge of interface description languages and a background in programming, and whether the underlying principles are not too challenging to adopt for both providers and client-developers novice to the field.

When it comes to the data consumed and produced by the APIs in the form of input and output, the results of the second study show that the description of the input parameters continues to be rather diverse and frequently underspecified (Section 6.4 point (5)) while XML, in particular, and JSON are becoming de-facto standard output formats (Section 6.4 point (6)).

2) The API input is transmitted via different parts of the HTTP message and not only in the form of parameters in the invocation URI.

This finding is quite important for developing a description model with wide coverage, which supports API invocation, since we can no longer assume that capturing the used parameters as part of the URI is sufficient.

3) The expected output format of the API is commonly requested as part of the input.

This results again in additional requirements for designing a Web API description model, since the output format has to be explicitly captured, instead of transparently handling it via content-negotiation.

The results of the second survey confirm the importance of authentication as part of supporting API use (Section 6.4 point (7)), since currently 80% of the APIs require some form of authentication. In addition, the landscape of used authentication credentials and approaches continues to be quite diverse and heterogeneous.

Finally, still most API documentations are characterised by some missing information that would be required for completing common service tasks and especially invocation (Section 6.4 point (8)). These usually include not providing the parameter datatypes and omitting the HTTP method. Even if currently the underspecification is not effecting the level of API reuse, despite that it represents an implementation overhead for the application developers, it is important to

give providers some initiative to offer complete documentation. A first step in this direction would be an API description model that includes all the characteristics for enabling API use.

In summary, we can conclude that currently Web API documentation is human-oriented and not meant for automated processing by Web service search engines or Web API composition and invocation frameworks. In addition, Web APIs proliferate without conforming to any standards or guidelines, relying on HTML webpages for giving details about the URI used for identifying endpoints, and HTTP for passing and receiving data, and completing the communication. However, the simplicity of the underlying technology stack is accompanied by heterogeneity and underspecification of the provided API documentation. As a result, the use of APIs requires extensive manual effort and the realised implementation solutions are usually API-specific and, therefore, rarely reusable. The two studies provide the basis for identifying common API characteristics and aligning these with the pieces of information, which are required for enabling main service tasks, such as discovery, composition and invocation, in order to develop a description model capable of supporting unified and more automated API use.

6.5 The Web API Survey System

This section briefly describes the Web API survey system, which was designed and implemented in order to ease the collection of data about different Web API features. More details on the system, including data model, design, implementation, setup, and user instructions are available in Appendix B.

The experience of the first study, where all the results were collected in a single spreadsheet, raised the need for a supporting tool that would speed up the analysis process and let the survey participant focus on the API documentation and not on how the collected inputs are stored and structured. The survey system is implemented as a Web application that can be easily configured and without redeployment be used to conduct different surveys in parallel. Furthermore, it was designed to ease the gathering of Web API details by enabling people from distributed locations to provide input, thus supporting the crowd-sourcing of the Web API analysis task.

Figure 6.6 shows the first pages of the user interface. Some of the API details are already displayed, based on the API's documentation in ProgrammableWeb, while others, such as the category, and whether the URL points to the API documentation, have to be filled in by the survey participant.

The survey system implements all API characteristics used as part of the second survey, however, it can be configured to show only a desired subset in order to collect information about a restricted number of features. For example, the system can be used to gather details related only to classification or only to authentication. The result is a flexible and customisable Web

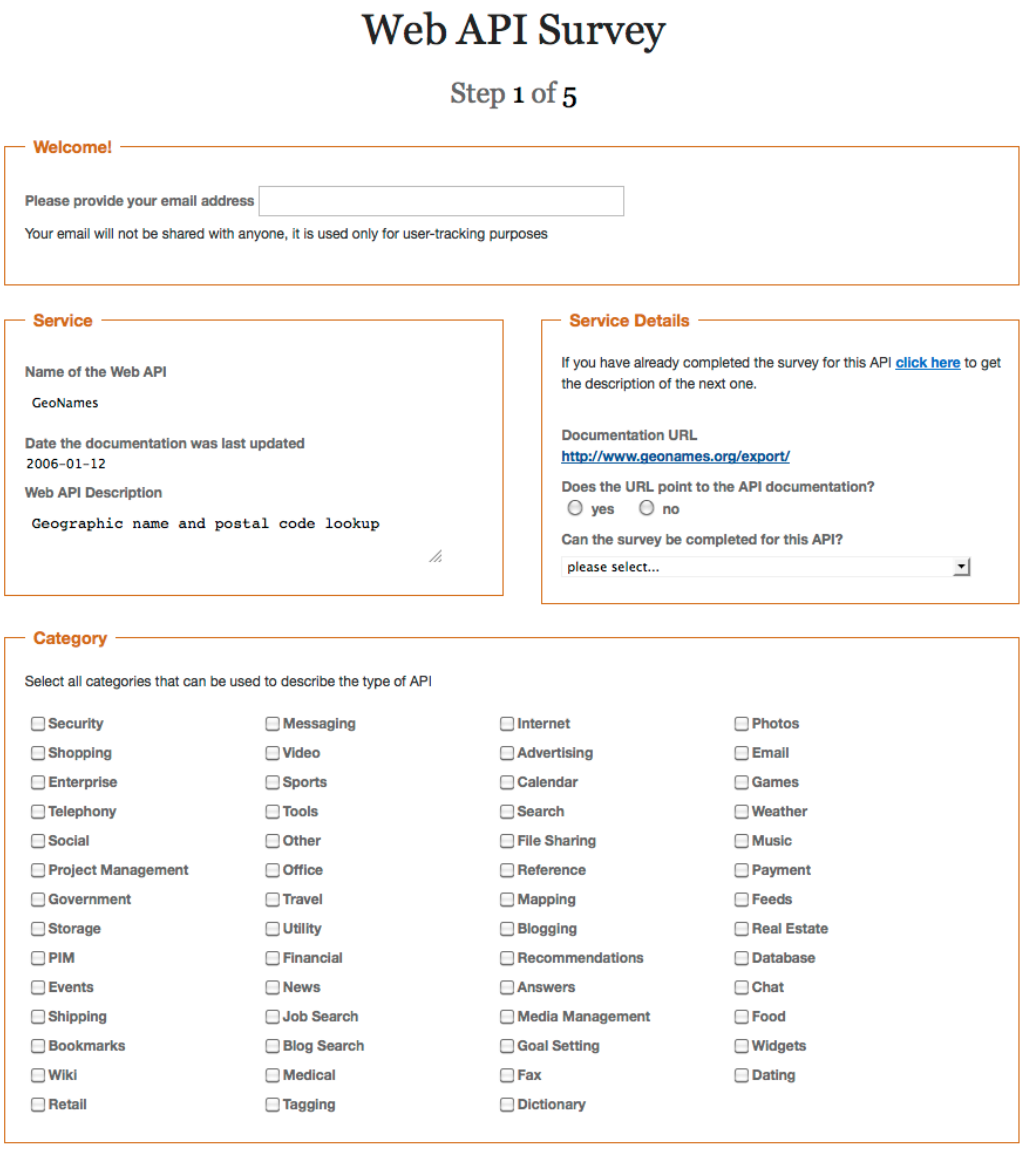


FIGURE 6.6: Web API Survey System - Form 1

application that can be distributed to different groups of participants in order to serve a variety of research interests.

6.6 Summary

Simply by browsing through the documentation of the most popular Web APIs, it becomes evident that the diversity in the used description forms and structure as well as the level of provided detail vary greatly from API to API. Therefore, before any significant progress and improvement can be made towards supporting and automating the use of Web APIs, we need to reach a deeper understanding of how APIs are developed and exposed, what kind of descriptions

are available, how they are represented and how rich these descriptions are. We contribute directly to this goal by providing two thorough studies on the current state of Web APIs based on investigating six groups of main features – general information, type of Web API, input details, output details, invocation details and complementary documentation. In addition, we provide a survey system that is customisable and can be used for a series of studies, investigating a variety of API characteristics. As a result, by using the collected data, we can better realise what the current difficulties are, which problems need to be addressed, and how should supporting mechanisms be devised.

In this sense, we show that currently Web APIs descriptions are human-oriented and not meant for supporting automated API processing. The results of the first study demonstrate that RESTful services are not the driving force behind the current Web API proliferation and that Web API descriptions are characterised by under-specification, where important information such as the datatype and the HTTP method is commonly missing. Instead, simplicity and the trend towards opening data are driving the evolution that results in the world of services on the Web being increasingly dominated by Web applications and APIs, which seem to be preferred over traditional Web services based on WSDL and SOAP. These initial observations are confirmed by the second study, which in addition also shows that the Web API environment is very dynamic with old APIs becoming no-longer available and new ones being offered. Moreover, by reflecting on the results of the initial survey, we were able to refine the analysed features towards gathering further details and deriving requirements for designing a description model capable of supporting more automated Web API use. Therefore, the impact of the API surveys is twofold. First, they determine the details that need to be captured by the description model, such as the different types of input parameters or in which part of the HTTP request they are transmitted. Second, they serve as input for decisions on whether certain features should be included or not and in what form, in order to ensure a greater coverage of the model, such as for example, taking a resource or operation-based approach towards describing Web APIs. As a result, the data gathered by the surveys serves as the basis for the evolutions and updates for the Minimal Service Model, and drove the definition of the Web API Grounding model and the Authentication model, presented in the following chapters.

In the following chapter we describe the Minimal Service Model and demonstrate how the results of the two surveys influenced some of its properties and shaped its design.

Chapter 7

Describing Web APIs

This chapter introduces the core service model, which on the one hand aims to support the description and modelling of Web APIs, and on the other hand enables the unified handling of APIs and RPC-oriented services, such as WSDL-based ones. In particular, it uses the data gathered by the two Web API studies, takes into consideration existing description approaches and presents a set of requirements for designing a model, capable of capturing the majority of the existing APIs. The result is the Minimal Service Model (MSM), which represents an operation-based approach towards describing APIs, thus supporting the description of operation-based services in general. The applicability of MSM is demonstrated based on a simple example and is evaluated in terms of its conformance with the postulated requirements and the overall coverage that it provides (see Chapter 11 for details on the evaluation).

7.1 Introduction

The results of the two surveys, introduced in the previous chapter, clearly demonstrate that currently the Web API landscape is very heterogeneous. Moreover, there is no common format or structure for describing APIs and as a result, they all present different characteristics and different levels of documentation depth and detail. These facts make the task of developing a common Web API description approach a very challenging one.

Our work is inspired by the gathered data, the analysis of work related to SWS and existing Web API description approaches in order to derive a unifying model used to describe and capture the characteristics of Web APIs. In particular, the so designed core model provides the basis for the unified handling of Web APIs and serves as the foundation for enabling the development of automated algorithms for completing common service tasks. In addition, extensions to the model are developed in order to give specific support for individual service tasks, such as invocation

and authentication. In summary, the here presented model enables the semantic description of Web APIs but also serves as the foundation for modelling, annotating and invoking APIs and operation-based services in general.

This chapter is structured as follows: Section 7.2 gives a discussion about current Web API approaches and points out some issues that need to be taken into consideration while defining the core service model. Section 7.3 gives our definition of a Web API, as used in the context of the here described work. Section 7.4 includes a set of requirements, which result directly from the two surveys and need to be satisfied by appropriate elements in the core service model. The Minimal Service Model (MSM), as well as a motivation of the design decisions and some examples, is provided in Section 7.5. Finally, Section 7.6 concludes the chapter.

7.2 Discussion

Despite the fact that Web API popularity has been increasing over the past 6-7 years, it is still often unclear what the main driving principles are and how they relate to traditional Web Services. This section aims to address precisely some of the misconceptions and grey areas, when it comes to defining what a Web API is. In particular, we discuss:

- The main differences between WSDL/SOAP-based services and Web APIs;
- The relationship between Web APIs, RESTful services and data endpoints;
- The relationship between implementation and interface definitions.

Currently, it is often unclear why Web APIs are more suitable than Web services for certain use cases and what the fundamental differences between the two approaches are [PZL08]. It is important to point out that Web APIs and Web services rely on different technology stacks and have different description forms – Web APIs have mostly human-oriented documentation such as text/ HTML given as part of webpages, while Web services have XML descriptions meant for machine interpretation, which can be accompanied by further textual documentation. Web services are guided by standards (such as the ones by OASIS[1] or IBM[2]) and specifications that clearly prescribe the form and structure of the descriptions, the format of the communication messages, the ways of providing additional information and meeting further requirements such as message security or quality of service. In contrast, Web APIs have evolved rather autonomously and it is up to the provider to decide what information to include in the documentation, how to implement the service, by applying REST or rather by using an RPC-approach (see Chapter 3 for details), and how to expose the data. In summary, Web services and Web APIs represent two completely

[1] `https://www.oasis-open.org/standards`
[2] `http://www.ibm.com/developerworks/webservices/standards/`

contrasting approaches towards technology evolution – standardisation vs. autonomous development.

However, the most important difference is not in the development approach but rather in the underlying technologies. Web services employ a quite complex stack, starting with WSDL and SOAP and building up to security, reliable messaging, transactions, WS-BPEL, etc. (see Chapter 3). They rely on SOAP messages that can be transported over HTTP and employ the generation of stubs and hubs, which enables the client-server communication. In contrast, Web APIs use mostly only natural language based documentation, which needs to be interpreted before a corresponding implementation can be realised that handles the HTTP-based message exchange. In essence, they adopt the fundamental Web technologies in order to enable decoupled programmable access to resources and pre-implemented functionality.

Another important differentiation is between Web APIs, RESTful services [RR07] and data endpoints. As we have demonstrated, by the results of the two studies (see Chapter 6), only a small percentage of the Web APIs are in fact RESTful. Still it is very common that the terms "RESTful services" and "Web APIs" are used interchangeably. This is misleading and inaccurate, and in such cases the RESTful services do not actually strictly conform to all REST principles but are referred to as "RESTful" because of the HTTP-based communication.

Furthermore, since Web APIs provide access to resource representations, sometimes it is unclear how they differ from data endpoints, such as SPARQL [PS08] endpoints, which in principle enable RDF data querying support over HTTP, or a simple Web service can enable access to server resources or a file directory over HTTP. Sometimes Web APIs provide access to the functionality of components and in other cases they expose resources. Therefore, it might be difficult to differentiate when an API, or a service in general, enables the direct access to data and when it actually does some computation in order to determine the data. In essence, it is up to the developer to explicitly define the Web API as a service and not as a data endpoint, and to implement it as such.

In addition, the actual processing logic is hidden behind the Web API or service interface. Therefore, we can only judge by the description of the interface, what the service actually provides. Still, only by the description, we cannot actually know if a Web API directly provides access to resources available on the server or if there is processing and computation required, which is not visible for the client application. This makes the differentiation between APIs and data endpoints less clear. In general, if there is only little or no processing required in order to retrieve some data, we can talk about dealing with a data-provisioning system. In the context of our work we focus on Web APIs in general, independently on whether they are more data-centric or rather provide access to computational functionality. The important distinction is rather the way of exposing the interface, in terms of an URI identified endpoint and direct HTTP messaging, with data payloads sent and received as part of the requests and responses.

7.3 Definition of a Web API

Based on the Web API discussion, the related work (see Section 3), and taking into account the data collected by the Web API surveys and the existing work related to creating Web API descriptions, we deduce a Web API definition, introduced in this section.

A **Web API**, as defined within the scope of this work, is an endpoint[3] that provides access to functionalities or resources in a programmable way[4] over the World Wide Web, i.e. via Web-related standards such as URIs and HTTP. The implementation of these resources or functionalities is not part of the Web API itself, which only provides an interface for accessing them. The endpoint is identified by a URI and the means of communication with it are specified in an interface definition. The interface definition[5] can be based directly on the REST principles, or be given in a description, which is in natural language or in a machine-interpretable format such as an XML-based language. The interface definition can be accompanied by additional documentation such as a webpage, workflow graphics, and sample messages. Web APIs should conform to the following characteristics:

1. Communication between the client and the Web API server is realised over HTTP, with the help of HTTP request and response messages.
2. The Web API exposes a set of operations, which can be the uniform set of REST operations or an arbitrary set of operations.
3. An operation is uniquely identified by the combination of the used HTTP method and an endpoint URI.
4. The Web API operations can have input data, which can be simple key-value pair or complex objects, and is passed as part of the HTTP request (either as part of the URI, the HTTP Body or the HTTP Header). However, there are Web APIs operations that require no input data.
5. The Web API operations have output data, which is transmitted in the Body of the HTTP response. The output data can be retrieved in different formats, which are usually machine-interpretable.

Web APIs rely on an interface, in order to enable the communication between pieces of software over the WWW. In the case of RESTful APIs the specification of the interface is implicit, determined by the REST principles. For the majority of the Web APIs, the interface specification is

[3] An *endpoint* in this context is a specific location for accessing a service, specified via URI, using a specific protocol.

[4] A way that is meant for machine consumption and interpretation, instead of, for instance, a human user.

[5] An *interface definition* determines the types of messages and the message exchange patterns, including any conditions implied by those messages.

done as part of an explicit description, which is usually in a human-oriented format – HTML/ text. The core service model aims to formally define the description of the interface and thus promote a common understanding, in order to facilitate reuse and compatibility of the produced solutions.

7.4 Requirements

In this section we focus on deriving the requirements for a core service model, capable of capturing common Web API characteristics and thus providing description support for the majority of the APIs. Furthermore, the requirements also reflect our goal to develop a model that enables the unified and automated handling of traditional WS and Web APIs. This will provide the foundation for the development of integrated solutions based on both using APIs and WS, and directly contributes towards achieving the vision of Open Services on the Web. In this context we use the data gathered by the two surveys and derive the following set of requirements:

- **R1: The model should be able to describe the majority of the APIs on the Web**. The aim is to provide a solution that can be adopted for the majority of the APIs, thus enabling the development of approaches with wide applicability and coverage. This includes also capturing common Web API characteristics.

This requirement is a crucial one, not only for MSM but also for the Web API Grounding Model and the Web API Authentication Model, since without sufficient coverage, the models would not really be applicable. In this context, we tried to support as many of the Web API characteristics identified through the Web API surveys as possible, however, still aiming to enable the easy creation of annotations. Trying to find a balance between the level of detail and the level of complexity, we aimed for a coverage of at least 80%. This number was determined by identifying that about two thirds of the APIs have interfaces based on operations. We aim to cover all operation-based Web APIs and, in addition to that, we want to support at least a half of the remaining APIs (resulting to a total of about 82%). As we demonstrate in the chapter on evaluation (see Chapter 11), we actually cover a larger percentage.

- **R2: The core service model should take an operation-based view**, as opposed a resource-based one. Since, the majority of the Web APIs are described in terms of operations and we aim to provide high coverage, we need to be able to capture the common features of as many APIs as possible. Furthermore, an operation-based view would enable the easier integration with WS and support the reuse of SWS approaches.
- **R3: The core service model should be able to describe input and output data of the APIs' operations**. Two very important elements of the Web API description are the

operations' inputs and outputs, which should be machine interpretable on syntactic level[6] and extendable with semantic information about the particular type of data. Furthermore, based on the collected statistics from the Web API surveys, we have determined that both the data transmitted to the server, as well as the data sent as part of the response, can be either in the form of simple key-value pairs or be complex objects. Therefore:

R3.1: The core service model should be able to describe individual operation parameters but also the complete inputs and outputs as a whole. This is an important requirement not only in the context of capturing key Web API features but also because the detailed description of the data payloads is crucial for supporting a more automated completion of tasks such as discovery, compositions and especially invocation. Here again, we aim to capture the parameters at the syntactic level and subsequently enable their enhancement with semantic metadata.

R3.2: The core service model should be able to describe optional and mandatory input parts. We discovered that a large percentage of the Web APIs use optional parameters. Therefore, we need to be able to individually describe the separate input parts, in addition to the complete input message as a whole. This requirement has a direct influence on the level of support that the model provides in terms of discovery, since depending on the available data, Web APIs that require some types of input can be found or not. Similarly, invocation would provide different results depending on whether an optional parameter with default values is used or only required parameter values are provided.

- **R4: The core service model should be able to describe the used HTTP method**. The HTTP method is crucial, especially in the context of invocation, since it specifies the way that the request is sent to the server.

- **R5: The core service model should be able to describe the endpoint URI**. Similarly to the HTTP method, the actual endpoint is of high importance, especially for the invocation, since it determines how to access the API.

In summary, the here listed requirements are based on common features that were defined as part of our analysis of the current state of Web APIs. Furthermore, they take into account compatibility with traditional WS definitions and basic task support that needs to be guaranteed through the specification of key elements of the model. In the following section, we describe in detail the resulting Minimal Service Model (MSM).

[6] In this context, we mean machine-oriented syntactic structuring of the operations' inputs and outputs.

7.5 Core Service Model

In this section we introduce the core service model. In particular, we motivate the design decisions that were made, when determining the individual model parts. We describe MSM and its characteristics in detail, and also provide some annotation examples.

7.5.1 Design Decisions

The purpose of the core service model is to provide unified handling and automation support for common service tasks for the majority of the APIs, and for Web services in general. As already mentioned, we aim to achieve this goal by using the data from the two Web API surveys and taking into consideration existing service description approaches. The process of defining the model was guided by a number of design decisions, which needed to be made before determining the individual elements. In particular, we follow a traditional ontology engineering process towards developing the Web API core model [SSS06, Ĺ99]. The used method is very much aligned with the phases of the ontology lifecycle [Gom98], undergoing specification, conceptualisation, formalisation and implementation. In this chapter we reflect on how we have undergone each of these phases, without talking about ontology engineering explicitly.

Therefore, the steps followed while developing the Web API description model are the following:

1. Based on the Web API analysis, we define an initial set of service elements, which need to be included in the API description. This includes considering existing models and their potential for being reused or extended.
2. We design a core service model capable of capturing common service properties.
3. We develop extensions to the core model for supporting the automation of specific service tasks, focusing in particular on invocation and authentication.

Guided by the results of the Web API study, we analysed the collected data and derived a core service ontology, which enables the annotation of common service properties as part of a semantic Web API description. The process of defining this ontology was guided by a number of competency questions and design principles.

First, we started by identifying the main service properties, which need to be captured and their specific characteristics. Relevant information in this respect is: *"What are the main service elements?"*, *"What are the relationships between them?"*. These questions can be refined: *"What are the operations of service X?"*, *"What are the inputs and outputs of operation X?"*, *"What*

are the error messages of operation X?", *"What are the different parts of input X?"*, *"What are the different parts of output X?"*, *"What are the relationships between the parts of input/output X?"*, and *"What are the different types of input/output parts?"*.

Each operation has an endpoint and an HTTP method that can be used to call it. We modelled this by reflecting on: *"What are the HTTP methods of operation X?"* and *"What are the endpoints/addresses for operation X?"*. In order to reuse existing ontologies we tried to link to already defined concepts instead of defining new ones – *"Are there ontologies that define these concepts?"* and *'Which exciting ontologies can be used to extend the model?"*.

In addition to the competency questions, used for identifying the information that needs to be captured by the core service model, we implemented some complementary requirements. In particular we followed the Principles for the Design of Ontologies [Gru95]:

- Clarity: To communicate the intended meaning of defined terms.
- Coherence: To use terms and rules that are consistent with definitions and inference results.
- Extensibility: To anticipate the use of the shared vocabulary.
- Minimal Encoding Bias: To be independent of the symbolic level.
- Minimal Ontological Commitments: To make as few claims as possible about the world.

These competency questions and the requirements ensure that the resulting ontology can capture the common service properties and their relationships, thus being able to model the majority of the Web APIs. The so designed ontology is not bound to any particular annotation formalism and, based on the principle of *Minimal Ontological Commitments*, it should be easy to interpret and use while generating annotations. In the next section we describe the Minimal Service Model in detail.

7.5.2 Minimal Service Model

The Minimal Service Model (MSM) is a simple RDF(S) ontology that supports the description of Web APIs, based on common service properties. It also aims to enable the reusability of existing SWS approaches by capturing the maximum common denominator between existing conceptual models for services. Additionally, as opposed to most SWS research to date, MSM aims to support both "traditional" Web services, as well as Web APIs with a procedural view on resources, so that they can be handled in a unified way.

As seen by the two Web API studies, currently documentation is given directly as part of a webpage, not conforming to any particular format or structure. Therefore, usually there is no

explicit description and client application developers have to read and interpret the documentation, identifying the individual service properties. Originally MSM was introduced together with hRESTS [KGV08], which aims to address the difficulty that the service structure is not automatically recognisable in the HTML documentation. In particular, hRESTS enables the marking of service properties including service, operations, inputs and outputs, HTTP methods and labels, by inserting tags within the HTML (see Chapter 3). MSM has been subsequently adjusted and updated to its current version, which not only enables the capturing of the structure of services (missing in the context of Web APIs) but also provides means for integrating heterogeneous services (i.e., WSDLs and Web APIs).

In particular, MSM, can be used in conjunction with WSMO-Lite in order to facilitate a common framework covering the largest common denominator of the most used SWS formalisms on the Web. WSMO-Lite provides support for capturing the main semantics of services (data model semantics, functional semantics, nonfunctional semantics, and behavioural semantics) through simple annotations using SAWSDL constructs. Therefore, MSM together with WSMO-Lite build the foundation for the provisioning of generic publication and discovery solutions that support not only Web APIs with textual documentation but also SAWSDL, WSMO-Lite, hRESTS/MicroWSMO, and OWL-S services [PLM+10].

The so developed services share a common conceptual model – MSM, and are directly integrated with existing Linked Data [BHBL09]. This integration serves both as a means to simplify the creation and management of Semantic Web Services through reuse, as well as it provides a new view over Semantic Web Services understood as a means to support the generation and processing of Linked Data. Therefore, MSM directly contributed towards achieving the vision of *Open Services on the Web*. Subsequent work around the Minimal Service Model is focused on supporting the invocation and authentication of Web APIs (see Chapter 8 and Chapter 9).

In the following we provide more details on the MSM ontology and its individual concepts and properties. The original MSM [KGV08] defined a Web API in terms of a *Service* that has a number of *Operations*, which have an *Input*, an *Output*, and *Faults*. However, the original MSM, which was used as a basis for SA-REST [SGL07] and MicroWSMO [KV08], fails to capture some significant parts of the descriptions that our survey highlights as necessary. This for instance concerns optional, default and mandatory parameters, which can have a crucial effect on discovery and invocation. In addition, it does not enable the direct description of the inputs/outputs, their parts, and the parts of their parts[7]. As a result, in the context of this thesis and guided by the analysis of the current state of Web APIs, MSM has been extended [PLM+10] to support some of the service characteristics identified by the survey.

[7]It enables it indirectly, via the conceptual definition of the annotated input/output, i.e. the conceptual definition contains the parts definitions.

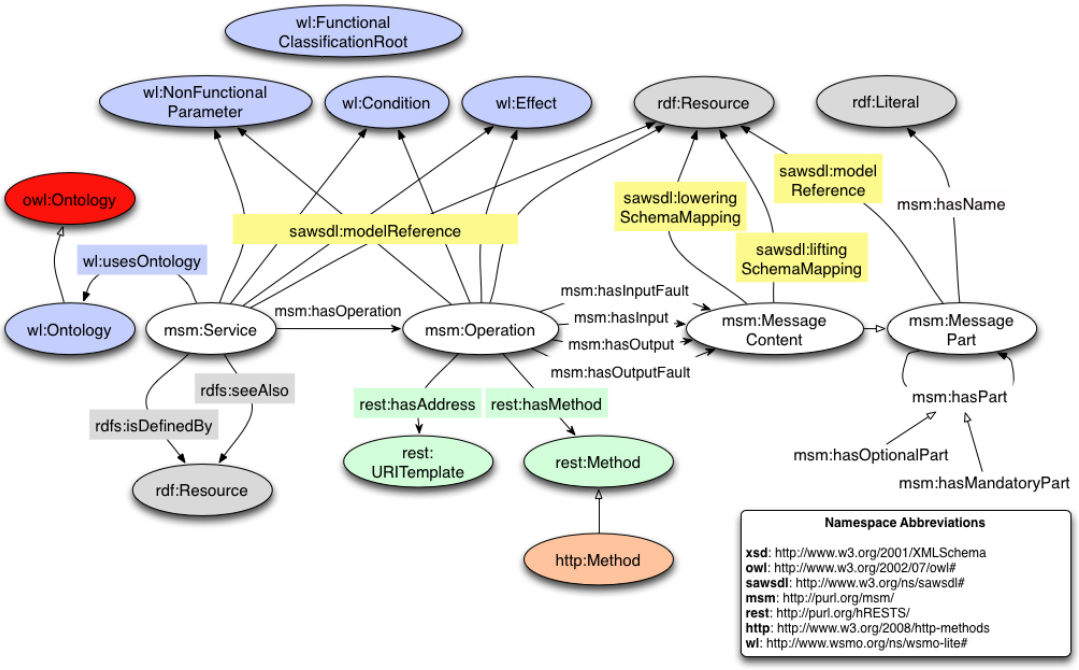


FIGURE 7.1: Minimal Service Model

MSM[8], in its current version, is visualised in Figure 7.1. MSM, denoted by the *msm* namespace, defines *Services* which have a number of *Operations*. *Operations* in turn have input, output and fault *MessageContent* descriptions. A *MessageContent* may be composed of *MessageParts*, which can be mandatory or optional. The intent of the message part mechanism is to support finer-grain processing of tasks, such as discovery, based on message parts and allowing to distinguish between mandatory and optional parts.

MSM[9] is used as the core structure for describing Web APIs, while supplementary models are defined for capturing details that are of particular relevance for the separate service tasks. As part of identifying extensions to MSM, work on supporting the automation of invocation [MPL+11] (see Chapter 8) and authentication [MPD+10b] (see Chapter 9) was conducted. Due to the fact that in our survey none of the analysed APIs had a machine-processable description in the form of a WSDL or WADL file, we accommodate the fact that providers publish details about services as text/HTML, and processing and automation support is based on subsequent enhancement of these with annotations.

Details about the evaluation of MSM in terms of requirements coverage and model coverage are given in Chapter 11.

[8]MSM ontology in RDF/XML available at `http://purl.org/msm/`
[9]Further details on MSM are available in the Appendix in Section A.1.

7.5.3 Using MSM to Create Semantic Web API Descriptions

In this section we provide a complete example that demonstrates how MSM can be used to create semantic Web API descriptions by annotating and processing existing documentation. In particular, MSM provides the conceptual view to a service, including its individual classes and properties. This conceptualisation can be used to create syntactic representations of MSM-based descriptions within HTML pages. This is realised by taking MSM ontological elements and mapping them on the level of text/HTML through the definition of microformat tags.

MSM can be used to directly create the corresponding service properties in RDF and complete the description, based on reading and interpreting the textual documentation. However, this requires some expert knowledge in the filed of semantics and APIs. Therefore, we advocate a bottom-up approach, where the HTML webpage that includes the Web API details is first syntactically structured by marking the service properties with the help of microformats. Subsequently, the identified service properties can be annotated with semantic entities. Finally, the so enhanced HTML can be used to extract the MSM-based representation of the API in RDF.

In this particular section we use hRESTS [KGV08] (see Chapter 3) as the means to represent the MSM conceptual view of a service within the HTML via MSM-hRESTS element mappings (see Table 7.1). In this way we enable the capturing of service properties on the syntactic level. Alternatively, RDFa is also an option for marking up HTML content, based on a service conceptualisation as defined in MSM. hRESTS, in general, enables the identification of operations, inputs, outputs, etc., by inserting HTML tags. We have extended the initially introduced hRESTS [KGV08] with further tags in order to be able to completely map to MSM (see Listing 7.2 for an example HTML documentation with hRESTS annotations). The full list of MSM-based hRESTS tags is given in the Appendix in Section A.1 and is discussed in the following section.

MSM only captures the structure (service, operations, inputs, outputs, etc.) but not the semantics of a service (semantics of the data model, functionalities, etc.). To this end, the structural descriptions are enriched with annotations to domain ontologies or existing rules by using pointers based on adopting the SAWSDL [FL07] approach (see Chapter 3). In particular, we enhance service properties with metadata, by using the *modelReference* link relation, which can be applied on any service property to point to semantic concepts identified by URIs (see Listing 7.3 for an example HTML documentation with *modelReference* annotations).

The annotation process for creating MSM Web API descriptions is as follows:

- We start with the HTML documentation of the API, without any annotation markup.
- The HTML is enhanced with hRESTS annotations that identify the individual service properties, in order to create the syntactic MSM-based structure of the service.

- Subsequently, we add metadata by using the SAWSDL *modelReference* link relation to point to semantic entities.
- Finally, a MSM-based RDF description can be extracted from the annotated HTML documentation.

The output of the annotation process is twofold – an HTML file containing hRESTS tags and *modelReference*s (see Listing 7.3 for an example), and an extracted RDF-based Web API description, which captures the API in terms of its MSM representation (see Listing 7.4 for an example).

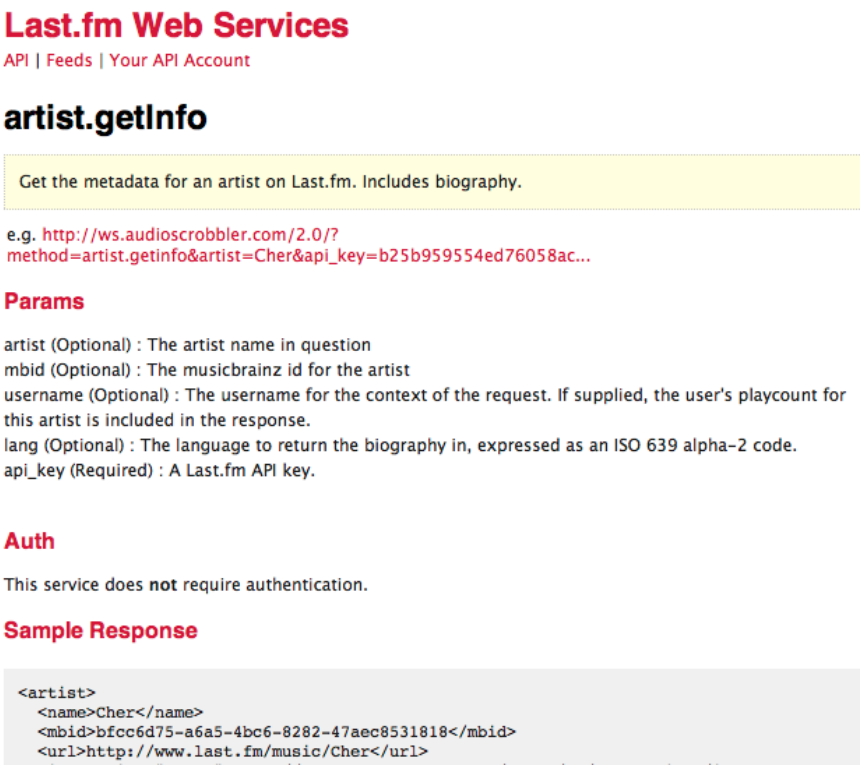


FIGURE 7.2: Last.fm HTML Example

In this section and throughout the following chapters, we use a running example, based on the Last.fm API. Last.fm[10] is an online community platform that collects and offers information about music artists, concerts, events, albums and songs. All the gathered data is also made available through a set of publicly exposed Web APIs[11], which enable the integration with further data sources or the building of applications.

Figure 7.2 shows a screenshot from the *artist.getInfo* operation of the Last.fm API. As can be seen, it contains a description of the input parameters, an example invocation address and a sample response. This operation can be used to retrieve details about a particular artist, given a name or the MusicBrainz Id ("mbid").

[10] http://www.last.fm
[11] http://www.last.fm/api

```
<h1 class="header"><a href="javascript:callProxy('http://www.last.fm/api')">Last.fm  Web Services</a></h1>
<h1>artist.getInfo</h1>

<div>Get the metadata for an artist  onLast.fm. Includes biography.</div>

<p>e.g. <a href="javascript:callProxy('http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&amp;
artist=Cher&amp;api_key=b25b959554ed76058ac220b7b2e0a026')">http://ws.audioscrobbler.com/2.0/?
method=artist.getinfo&amp;artist=Cher&amp;api_key=b25b959554ed76058ac...</a></p>

<div><h2>Params</h2>
<span>artist</span> (Optional) : The artist  name in question<br>
<span>mbid</span> (Optional) : The musicbrainz id for the artist<br>
<span>username</span> (Optional) : The username for the context of the request. If supplied, the user's
playcount for this  artist  is  included in  the  response.<br>
<span>lang</span> (Optional) : The language to return the biography in, expressed as an ISO 639 code.<br>
<span>api_key</span> (Required) : A Last.fm API key.</div><br>

<h2>Sample Response</h2>
<div id="sample">
<pre>
&lt;artist&gt;
  &lt;name&gt;Cher&lt;/name&gt;
  &lt;mbid&gt;bfcc6d75−a6a5−4bc6−8282−47aec8531818&lt;/mbid&gt;
  &lt;url&gt;http://www.last.fm/music/Cher&lt;/url&gt;
  ...
 &lt;/artist&gt;</pre></div>
```

LISTING 7.1: Example of HTML Description without Annotations

Listing 7.1 contains a simplified version of the HTML source of the *artist.getInfo* operation. We use this short HTML snippet in order to demonstrate how hRESTS, based on MSM element mappings, can be used to mark the individual service properties within the description.

7.5.3.1 Syntactic Structuring of Web API Documentation

Following our bottom-up approach, we show how the HTML documentation can be enhanced with hRESTS tags, which mark the individual service properties. Listing 7.2 includes the same operation webpage source, this time with the inserted HTML tags. As can be seen, the service body is marked with the *class="service"* and is assigned and Id – *id="LastFMService"*. Similarly, the part of the HTML that describes the operation is also marked (*class="operation" id="ArtistGetInfo"*). The same is done with the input, output and the address. The result is an HTML file, which contains a syntactically structured Web API documentation, based on hRESTS tags mapped to the conceptual representation of a MSM-based service.

As can be seen the *class="service"* tag marks the complete part of the HTML, which describes the different service properties. This tag is supposed to be placed in such as way as to encompass all the content related to the API. Within the *service* tag, *class="operation"* tags can be placed, which in turn can include *class="input"* and *class="output"*. The operation element also includes a *class="address"*, while the *input*, includes two parameters – an optional and a mandatory one. The nesting of the HTML tags reflects the relationship between the service

properties in an implicit way. For example, the fact that the input is defined within the operation tag means that it belongs to this particular operation (i.e. the *ArtistGetInfo* operation has *ArtistGetInfoInput*, which in turn has *artist* and *api_key*).

```
<div class="service" id="LastFMService">
<h1 class="header"><a href="javascript:callProxy('http://www.last.fm/api')">Last.fm  Web Services</a></h1>
<div class="operation" id="ArtistGetInfo"><h1><span class="label" id="label1">artist.getInfo</span></h1>

<div>Get the metadata for an artist  onLast.fm. Includes biography.</div>

<p>e.g. <a href="javascript:callProxy('http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&amp;
artist=Cher&amp;api_key=b25b959554ed76058ac220b7b2e0a026')"><span class="address" id="address1">
http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&amp;
artist=Cher&amp;api_key=b25b959554ed76058ac...</a></span></p>

<div class="input" id="ArtistGetInfoInput">
<div><h2>Params</h2>
<span id="artist" class="parameter">artist</span> (Optional)  :  The artist  name in question<br>
<span>mbid</span> (Optional) : The musicbrainz id for the artist<br>
<span>username</span> (Optional) : The username for the context of the request. If supplied, the user's
playcount for this  artist  is  included in  the response.<br>
<span>lang</span> (Optional) : The language to return the biography in, expressed as an ISO 639 code.<br>
<span id="api_key" class="parameter-mandatory">api_key</span> (Required) : A Last.fm API key.</div><br>
</div>
<h2>Sample Response</h2>
<div id="sample">
<div class="output" id="ArtistGetInfoOutput">
<pre>
&lt;artist&gt;
  &lt;name&gt;Cher&lt;/name&gt;
  &lt;mbid&gt;bfcc6d75−a6a5−4bc6−8282−47aec8531818&lt;/mbid&gt;
  &lt;url&gt;http://www.last.fm/music/Cher&lt;/url&gt;
  ...
 &lt;/artist&gt;</pre></div></div></div></div>
```

LISTING 7.2: Example of HTML Description with Annotations

This becomes even more evident when looking at the mappings that we defined between MSM and hRESTS, shown in Table 7.1. The properties from MSM that express the relationship from one concept to the other are reflected in hRESTS indirectly through the nesting structure, where the parent element *has* a child element with particular attributes. As already mentioned, hRESTS is only one possible way of using the MSM-based conceptualisation of an API and applying it on the syntactical level for structuring API documentation and making the contained service properties explicit.

The only new hRESTS elements that were introduced are the *parameter* and *parameter-mandatory*. The *parameter* is used to identify individual parts of the input and output, while the *parameter-mandatory* is used to mark required input parts. Some MSM properties and concepts are not directly mapped to hRESTS, including the ones related to faults. In this case, the corresponding elements have to be created directly as part of the semantic descriptions and instantiating MSM, without following the bottom-up approach of annotating the HTML documentation and using the annotations to extract an MSM-based description. This was intentionally done in order to stick to the pre-established format and ensure backward compatibility. The *rdfs:isDefinedBy* and *rdfs:seeAlso* properties are automatically generated, pointing to the

MSM Property/Concept	**hRESTS Element**
msm:Service	*class* attribute + *'service'* as a value
msm:hasOperation + msm:Operation	*class* attribute + *'operation'* as a value
msm:hasInput + msm:MessageContent	*class* attribute + *'input'* as a value
msm:hasOutput + msm:MessageContent	*class* attribute + *'output'* as a value
msm:hasPart + msm:MessagePart	*class* attribute + *'parameter'* as a value
msm:hasOptionalPart + msm:MessagePart	*class* attribute + *'parameter'* as a value
msm:hasMandatoryPart + msm:MessagePart	*class* attribute + *'parameter-mandatory'* as a value
msm:hasInputFault + msm:MessageContent	not mapped to hRESTS
msm:hasOutputFault + msm:MessageContent	not mapped to hRESTS
rest:hasAddress + rest:URITemplate	*class* attribute + *'address'* as a value
rest:hasMethod + rest:Method	*class* attribute + *'method'* as a value
msm:hasName + rdf:Literal	*class* attribute + *'label'* as a value
rdfs:isDefinedBy + rdf:Resource	not mapped to hRESTS
rdfs:seeAlso + rdf:Resource	not mapped to hRESTS
sawsdl:modelReference + rdf:Resource	*rel* attribute + *'model'* as a value + *href* to linked URI

TABLE 7.1: Mapping MSM to hRESTS/MicroWSMO Elements

original HTML documentation and to the annotated HTML documentation, correspondingly. Furthermore, the *rest:hasAddress* and *rest:hasMethod* properties are handled in more detail in the following chapter, which focuses on Web API invocation.

Indeed, it is up to the annotator to decide, which sections of the documentation is marked up and captured as part of the semantic description. In our example, the optional parameters *mbid*, *username*, *lang* are not identified via hRESTS tags, but they could have been. The annotator determines, which elements should be included in the syntactical structuring of the documentation and which should be avoided. Naturally, he/she has to take into account that the resulting description should still included all details required for supporting common tasks, such as discovery and invocation.

7.5.3.2 Enhancing Web API Documentation with Semantics

Once the HTML documentation is marked up with hRESTS and the syntactical structuring is completed, the so identified service properties can be enhanced with semantic annotations by linking the particular service elements to semantic entities (i.e. their URIs). As already mentioned, for this purpose we use SAWSDL *modelReference*s for pointing to semantic concepts identified by URIs.

As can be seen in Listing 7.3, the *service* element is enhanced with a *rel="model"* attribute-value pair and an *href* pointing to the linked semantic entity, in this case a class within a service categorisation ontology. The *rel="model"* represents the *modelReference* mapping to hRESTS and the added link describes the type of functionality that the service provides, in this case a music service.

```
1  <div class="service" id="LastFMService"
   rel="model" href="http://www.service-finder.eu/ontologies/ServiceCategories#Music">
2  <h1 class="header"><a href="javascript:callProxy('http://www.last.fm/api')">Last.fm  Web Services</a></h1>
3  <div class="operation" id="ArtistGetInfo"><h1><span class="label" id="label1">artist.getInfo</span></h1>
4
5  <div>Get the metadata for an artist  onLast.fm. Includes biography.</div>
6
7  <p>e.g. <a href="javascript:callProxy('http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&amp;
8  artist=Cher&amp;api_key=b25b959554ed76058ac220b7b2e0a026')"><span class="address" id="address1">
9  http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&amp;
10 artist=Cher&amp;api_key=b25b959554ed76058ac...</a></span></p>
11
12 <div class="input"  id="ArtistGetInfoInput">
13 <div><h2>Params</h2>
14 <span id="artist" class="parameter" rel="model" href="http://musicbrainz.org/mm/mm-2.1#Artist">
15 artist</span> (Optional) :  The artist  name in question<br>
16 <span>mbid</span> (Optional) : The musicbrainz id for the artist<br>
17 <span>username</span> (Optional) : The username for the context of the request. If supplied, the user's
18 playcount for this  artist  is  included in  the  response.<br>
19 <span>lang</span> (Optional) : The language to return the biography in, expressed as an ISO 639 code.<br>
20 <span id="api_key" class="parameter-mandatory" rel="model"
   href="http://purl.oclc.org/NET/WebApiAuthentication#API_Key">api_key</span>
21 (Required) : A Last.fm API key.</div><br></div>
22
23 <h2>Sample Response</h2>
24 <div id="sample">
25 <div class="output" id="ArtistGetInfoOutput">
26 <pre>
27 &lt;artist&gt;
28   &lt;name&gt;Cher&lt;/name&gt;
29   &lt;mbid&gt;bfcc6d75−a6a5−4bc6−8282−47aec8531818&lt;/mbid&gt;
30   &lt;url&gt;http://www.last.fm/music/Cher&lt;/url&gt;
31   ...
32  &lt;/artist&gt;</pre></div></div></div></div>
```

LISTING 7.3: Example of HTML Description with Semantic Annotations

Furthermore, the two parameters *artist* and *api_key* are also enhanced with links that describe the input elements as an artist from the MusicBrainz[12] ontology and as an API key from the Web API Authentication ontology (see Chapter 9), correspondingly. In this way, the syntactically structured documentation can be enhanced with links to semantic entities.

7.5.3.3 MSM-based Semantic Web API Descriptions

In the final step of our bottom-up approach we demonstrate how the annotated HTML documentation can be processed in order to derive an MSM-based representation of the API. In particular, the HTML visualised in Listing 7.3 can be used to automatically extract a semantic Web API description in RDF. This can be done with the help of a simple XSLT transformation [Kay07, Bre09] that exploits the hRESTS-MSM element mappings and transforms the corresponding HTML parts to RDF statements.

Listing 7.4 shows the result of the transformation. As can be seen the elements defined by the *msm* namespace are *msm:Service*, *msm:Operation*, *msm:hasInput*, *msm:hasOutput*, *msm:MessageContent* and *msm:MessagePart*. These are derived directly based on the annotated HTML, by

[12]MusicBrainz – the open music encyclopaedia, `http://musicbrainz.org`

transforming the tags to the corresponding RDF elements. Furthermore, the two *msm:MessageParts* have attached a *sawsdl:modelReference*, adding further details. For example, the *artist* is described with the *Artist* class from the MusicBrainz ontology.

```
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix msm: <http://purl.org/msm#>.
@prefix rest: <http://purl.org/hRESTS#>.

:LastFMService a msm:Service;
   sawsdl:modelReference <http://www.service−finder.eu/ontologies/ServiceCategories#Music>;
   msm:hasOperation :ArtistGetInfo.

:ArtistGetInfo  a msm:Operation;
   msm:hasInput :ArtistGetInfoInput;
   msm:hasOutput :ArtistGetInfoOutput;
   rest:hasAddress "http://ws.audioscrobbler.com/2.0/?method=artist.getinfo&artist={artist}&api_key={api_key}".

:ArtistGetInfoInput  a msm:MessageContent.
   msm:hasPart :artist;
   msm:hasMandatoryPart :api_key.

:artist  a msm:MessagePart;
   sawsdl:modelReference <http://musicbrainz.org/mm/mm−2.1#Artist>.

:api_key a msm:MessagePart;
   sawsdl:modelReference <http://purl.oclc.org/NET/WebApiAuthentication#API_Keys>.

:ArtistGetInfoOutput a msm:MessageContent.
```

LISTING 7.4: Example of RDF-based Web API Description

The derived semantic description can be used as the basis for making further statements about the individual service properties. For example, we can add fault messages or further define the *msm:MessageParts* by specifying whether they are required or optional. Listing 7.5 shows how the input can be modified to contain two optional parameters and one mandatory parameter– *msm:hasOptionalPart* and *msm:hasMandatoryPart*.

```
@prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix msm: <http://purl.org/msm#>.

:ArtistGetInfoInput  a msm:MessageContent ;
  msm:hasOptionalPart :artist ;
  msm:hasOptionalPart :mbid ;
  msm:hasMandatoryPart :api_key .

:artist  a msm:MessagePart ;
   sawsdl:modelReference <http://musicbrainz.org/mm/mm−2.1#Artist> .
:mbid a msm:MessagePart ;
   sawsdl:modelReference <http://musicbrainz.org/mm/mm−2.1#ArtistId> .
:api_key a msm:MessagePart ;
   sawsdl:modelReference <http://purl.oclc.org/NET/WebApiAuthentication#API_Keys> .
```

LISTING 7.5: Example of Optional and Mandatory Parts

7.5.3.4 Describing Resource-Based APIs with MSM

MSM is based on defining an API in terms of operations, however, it is not limited to covering only operation-based documentation or only RPC-oriented interfaces. In this section we describe how resource-based APIs can be captured with MSM in order to create a semantic Web API description. Since resource-oriented APIs do not have operations, a key part in the process is deriving the operation by combing the resource and the used HTTP method. For instance, having a resource *News* that can be used with HTTP GET will result in a *getNews* operation.

It is important to point out that this way of deriving the operation is not limited to resource-based APIs that have a documentation, which provides details about the API characteristics. It can also be applied to RESTful services, without documentation, where the annotator would have to retrieve the list of exposed resources (similarly to retrieving the documentation, if there is a webpage describing the API) and define the operations in MSM, by combining each of the resources with the applicable HTTP methods. The inputs and outputs are determined in the same way as operation-based APIs, with the exception that the HTTP GET method has as output the resource itself, which might not be explicitly specified. We illustrate this consideration and the process of creating a MSM-based description via a detailed example.

Listing 7.6 gives part of the documentation of the BOX[13] data storage and management API. The BOX API gives access to the content management features, which are available through the web app for storing, managing and sharing personal documents online. In particular, we look at the *comment* resource and the part of the interface that enables the adding of new comments to an item[14]. The available resource is *comment*, while the set of properties that a comment can have are – *item*, *type*, *id*, and *message*. All of the comment properties are required.

```
1  HTTP Method: POST
2
3  Resource: /comments
4  Response: The new comment object is returned. Errors may occur if the item id is invalid , the item type is
5   invalid /unsupported, or if the user does not have access to the item being commented on.
6
7  Description: Used to add a comment by the user to a specific file or comment (i.e. as a reply comment).
8
9  Comment Properties:
10 Name        Value   isRequired   Description
11 item        object  required     The item that this comment will be placed on.
12 type        string  required     The type of the item that this comment will be placed on.
13                                  Can be file or comment.
14 id          string  required     The id of the item that this comment will be placed on.
15 message     string  required     The text body of the comment
```

LISTING 7.6: Example of a Resource-based Web API - Add a Comment to an Item

[13] BOX for online file sharing, `https://www.box.com`

[14] `http://developers.box.com/docs/#comments-add-a-comment-to-an-item`

Listing 7.7 shows a simplified example HTTP request, where a new comment is created. The comment resource properties are formatted in JSON and the used HTTP method is POST. Therefore, based on the described approach for deriving the operation, it will be defined by combining POST and *comment* to result in 'postComment'. If in some cases, the label does not really capture the semantics of the activity, the MSM-based description can still be enhanced with, for instance, a classification annotation that states that this is an API for adding comments.

```
1 curl https://api.box.com/2.0/comments \
2 -H "Authorization: Bearer ACCESS_TOKEN" \
3 -d '{"item": {"type": "file", "id": "FILE_ID"}, "message": "YOUR_MESSAGE"}' \
4 -X POST
```

LISTING 7.7: Resource-based Web API - Example Request for HTTP POST

Listing 7.8 shows a simplified example response, with the confirmation that the resource has been created (Line 1) and a pointer to where it has been created (Line 2).

```
1  Response: 201 CREATED
2  Location:https://api.box.com/2.0/comments/191969
3
4  {
5      "type": "comment",
6      "id": "191969",
7      "is_reply_comment": false,
8      "message": "These tigers are cool!",
9      "created_by": {
10         "type": "user",
11         "id": "17738362",
12         "name": "sean rose",
13         "login": "sean@box.com"
14     },
15     "created_at": "2012-12-12T11:25:01-08:00",
16     "item": {
17         "id": "5000948880",
18         "type": "file"
19     },
20     "modified_at": "2012-12-12T11:25:01-08:00"
21 }
```

LISTING 7.8: Resource-based Web API - Example Response for HTTP POST

Based on this example, we can use the provided details in order to create the following semantic Web API description. Listing 7.9 describes a service with a *postComment* operation. The operation has a method *POST*, an *input* and an *output* (Lines 14-16). The *input* consists of two parts, both of which are mandatory (Line 18-20). The first message part contains further parts, all of which are again mandatory (Lines 25-27). As exemplified, it is actually very easy to create the description.

```
1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
3 @prefix msm: <http://purl.org/msm#> .
4 @prefix rest: <http://purl.org/hRESTS#> .
5
```

```
:service1 rdf:type  msm:Service ;
    rdfs:isDefinedBy "http://developers.box.com/docs/#comments-add-a-comment-to-an-item" ;
    rest:hasAddress "https://api.box.com/2.0"  ;
    msm:hasOperation :postComment .

:postComment rdf:type msm:Operation ;
    rdfs:label  "Add a Comment to an Item" ;
    rest:hasAddress "/comments" ;
    rest:hasMethod "POST" ;
    msm:hasInput input ;
    msm:hasOutput output .

:input  rdf:type  msm:MessageContent ;
    msm:hasMandatoryPart  item ;
    msm:hasMandatoryPart message .

:output rdf:type  msm:MessageContent ;
    msm:hasPart comment .

:item rdf:type  msm:MessagePart ;
    msm:hasMandatoryPart :type ;
    msm:hasMandatoryPart :id .
:message rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_Body> .

:type rdf:type  msm:MessagePart ;
    sawsdl:modelReference <http://www.agls.gov.au/rdf/documentType> .
:id  rdf:type  msm:MessagePart ;
    sawsdl:modelReference <http://purl.org/atom/ns#id> .

:comment rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://www.w3.org/2001/12/replyType#Comment> .
```

LISTING 7.9: Example Semantic Web API Description - Add a Comment to an Item

We also provide an example based on using the HTTP GET method. Listing 7.10 shows a short documentation of the *comment* resource and how it can be used with HTTP GET.

```
HTTP Method: GET

Resource: /comments/{comment id}
Response: A full comment object is returned is the ID is  valid  and if  the user has access to the comment.

Description: Used to retrieve  the  message and metadata about a specific comment.
Information about the user who created the comment is also included.

Request Body Attributes: None are accepted
```

LISTING 7.10: Example of a Resource-based Web API - Get a Comment

Listing 7.11 shows sample HTTP request, where a comment is retrieved. As stated in the description, there are no input parameters passed in the HTTP body and the particular comment is directly requested over the URI and the *COMMENT_ID*, which would have an actual id value (similarly to the *ACCESS_TOKEN*). The used HTTP method is GET on the *comment* resource. Therefore, based on the described approach for deriving the operation, the resulting operation is 'getComment'.

```
curl  https://api.box.com/2.0/comments/COMMENT_ID
-H "Authorization: Bearer ACCESS_TOKEN"
```

LISTING 7.11: Resource-based Web API - Example Request for HTTP GET

Listing 7.12 shows a simplified example response, containing the complete comment message and related metadata.

```
{
    "type": "comment",
    "id": "191969",
    "is_reply_comment": false,
    "message": "These tigers are cool!",
    "created_by": {
        "type": "user",
        "id": "17738362",
        "name": "sean rose",
        "login": "sean@box.com"
    },
    "created_at": "2012−12−12T11:25:01−08:00",
    "item": {
        "id": "5000948880",
        "type": "file"
    },
    "modified_at": "2012−12−12T11:25:01−08:00"
}
```

LISTING 7.12: Resource-based Web API - Example Response for HTTP GET

Listing 7.13 gives the semantic Web API description to retrieving the *comment* resource with the HTTP GET method, based on the brief documentation in Listing 7.10. The service has an operation *getComment* with an *input* and an *output* (Lines 11-15). The *input* consists only of the comment id (Lines 17-18), which is sent directly as part of the URI, specifying the value in the address URI pattern (/comments/{comment id}).

```
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#> .
@prefix msm: <http://purl.org/msm#> .
@prefix rest: <http://purl.org/hRESTS#> .

:service1 rdf:type msm:Service ;
    rdfs:isDefinedBy "http://developers.box.com/docs/#comments−get−information−about−a−comment" ;
    rest:hasAddress "https://api.box.com/2.0" ;
    msm:hasOperation :getComment .

:getComment rdf:type msm:Operation ;
    rest:hasAddress "/comments/{comment id}" ;
    rest:hasMethod "GET" ;
    msm:hasInput input ;
    msm:hasOutput output .

:input rdf:type msm:MessageContent ;
    msm:hasMandatoryPart id .

:output rdf:type msm:MessageContent ;
    msm:hasPart comment .

:id rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://purl.org/atom/ns#id> .

:comment rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://www.w3.org/2001/12/replyType#Comment> .
```

LISTING 7.13: Example Semantic Web API Description - Get a Comment

Finally, the operation has one output, which is the actual comment containing all the comment details as well (as opposed to only the comment id). Further examples are included in the Appendix, in Section A.1.

In summary, independently whether creating the semantic Web API descriptions directly or by following the here presented bottom-up approach, MSM is a conceptualisation of a Web API, which can either be propagated to the syntactic level by creating corresponding mappings to the used annotation vocabulary (in our case hRESTS) or can be used to create MSM-based API representations. Due to the way, in which it was developed – by using the data gathered by the two Web API studies, taking into consideration existing description approaches and conforming to a set of requirements, MSM supports both the description and modelling of Web APIs, as well as the unified handling of APIs and RPC-oriented services, such as WSDL-based ones. We also demonstrate the direct applicability of MSM. In particular, it is used as the Web API model in SWEET (see Chapter 10), where it serves as the basis for making annotations on top of HTML documentation. In essence, SWEET is an annotation tool in the form of a Web application, which enables the creation of semantic Web API descriptions based on MSM by hiding the formalism complexities from the user behind a simple graphical interface. It enables the marking of service properties within the HTML, by simply selecting the corresponding part of the webpage and clicking on the element that should be associated with it.

7.6 Summary

The main contributions introduced in this chapter are twofold. First we give a definition of what a Web API is, as used in the context of this thesis. Second, we describe the core service model, which serves as the basis for capturing common API characteristics and providing a unified view over Web APIs and operation-based services in general, thus laying the foundation for developing solutions for higher levels of task automation. The core model is realised through MSM in the form of a simple RDFS ontology, containing four main classes – *Service*, *Operation*, *MessageContent* and *MessageParts*. As a result, it enables capturing the main service elements, which can be used to add further semantic annotations, for example, through the inclusion of *modelReferences*. Furthermore, MSM can be used in conjunction with WSMO-Lite for giving specific semantics to the annotations, or with the Web API Grounding Model to support invocation, as described in the next chapter.

Chapter 8

Supporting the Automated Web API Invocation

This chapter introduces the extensions to the core service model that were especially derived in order to better support invoking Web APIs automatically. The extensions consist of properties that enable capturing the details relevant for the construction and processing of the HTTP messages. The Web API descriptions, created with the help of the *Web API Grounding Model*, can be automatically processed by the invocation engine *OmniVoke*, without the need for customising or manually implementing a custom tailored client capable of calling the API. It is important to point out that in this chapter we focus solely on service properties required for constructing the HTTP request and processing the HTTP response, assuming that authentication is handled separately. We handle the modelling of commonly used authentication approaches in the following chapter, advocating a solution that is based on modularity and extensibility.

8.1 Introduction

Web service *invocation* is concerned with the actual calling of a Web service operation, including preparing the request message, and receiving and handling the response message [Cer02, PDS10]. We differentiate between *invocation* and *execution* [Cer02] by considering invocation as the task that needs to be completed by the client, while execution is performed solely as a processing step on the Web service server. Since our approach is based on a semantic model for describing APIs, we also consider the required groundings, i.e. liftings and lowerings, that need to be defined in order to be able to map semantically described messages to actual low level operations, to required data payload and to protocol messages of the Web APIs. The need for taking grounding into account arises from the fact that traditional Semantic Web Services and semantic Web APIs alike are essentially handled at the semantic level, which is sufficient

for tasks such as discovery and composition. However, the completion of invocation requires specific details, which are not available as part of the semantic descriptions.

Web API invocation is the main task that determines the successful usage of individual APIs, as well as applications built on top of them. In the context of the growing number of available Web APIs during the past few years, invocation plays a key role, since it can lead to a bottleneck in scalability, when it comes to developing implementation solutions and clients. In particular, as of now, the invocation of the majority of the Web APIs requires the manual implementation of custom-tailored clients for each individual API. This is mainly due to the following reasons:

- Heterogeneity of the Web APIs' idiosyncrasies, technical characteristics and messaging formats (for example, different approaches to implementing the interfaces – based on resources or operations);
- Underspecification of the interfaces (for example, missing HTTP method or missing datatype definition for the input parameters);
- Mixing of data-payload details with properties determining the implementation solution (for example, determining the output format – XML, JSON, etc., via an input parameter).

These issues can be directly observed in our analysis of the current state of Web APIs presented in Chapter 6. Despite the fact that considerable advances have been made towards providing support for enriching the HTML documentation of Web APIs with semantic annotations and exploiting these annotations for the identification and advanced discovery of Web APIs ([SGL07, KGV08, MPD09a, PLM+10]), supporting the automated invocation of Web APIs is yet to be addressed comprehensively. This is, in particular, due to the fact that automated service invocation has not been prominently addressed by semantic Web service research because the invocation of "traditional" Web services is directly enabled through WSDL. Invocation done on the level of WSDL and SOAP is well supported by a variety of frameworks and implementations (see Chapter 4). Invocation based on semantically annotated WSDL files is handled with the help of lifting and lowering transformations and the main concern in this case is the data transformation. In contrast, supporting the invocation of Web APIs generically presents a number of outstanding and somewhat unexpected challenges, resulting from the lack of a generally established interface description language (IDL), as described in more detail in Section 8.3.

This chapter is structured as follows: Section 8.2 presents an example that demonstrates the difficulties of performing Web API invocation. In Section 8.3 we systematically derive the requirements for designing a model capable of supporting automated Web API invocation and we present the resulting Web API Grounding Model in Section 8.4. Section 8.5 presents the invocation engine, while Section 8.6 completes the chapter by summarising the presented solution.

8.2 Motivating Example

In this section we present a simple example based on the previously introduced Last.fm[1] API. The example lets us demonstrate some of the main challenges related to invocation. Therefore, we use it throughout this chapter for illustrating the annotation approach and showing the resulting descriptions.

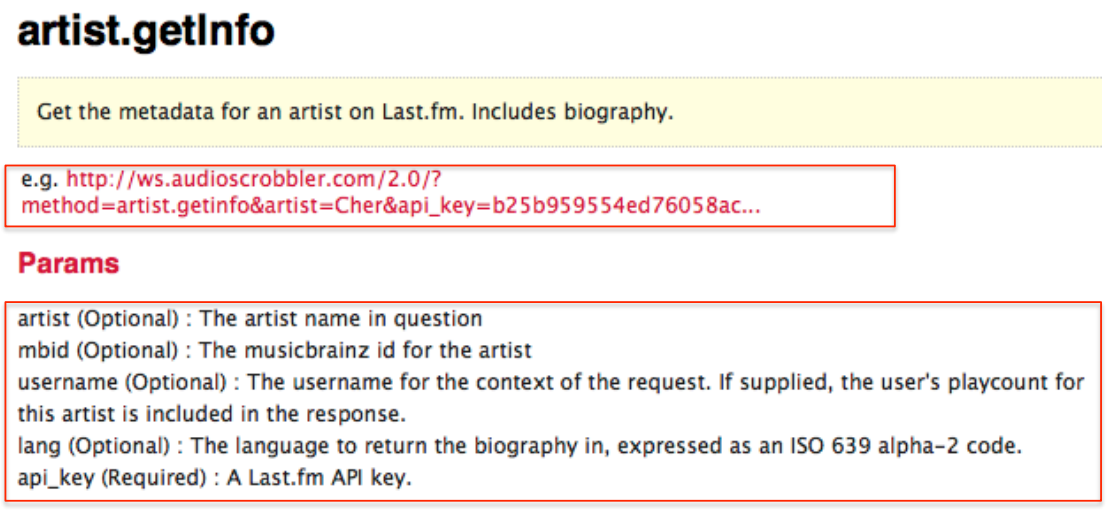


FIGURE 8.1: Extract from the Last.fm API

Figure 8.1 shows the HTML description of an operation for getting the details for a particular artist. This short description represents the only documentation available about the operation, implicating that developers have to manually go through it and determine the used parameters and the way of passing the input in an actual call. At first glance, it is not directly apparent that the provided description is missing key information, necessary for invoking the *artist.getInfo* operation. First, the HTTP method to be used is not explicitly stated but it has to be assumed that it is GET. As already pointed out in Chapter 6, the HTTP method is missing in 40% of the APIs [MPD10a]. Second, there is no information about the datatype of the inputs (as is the case for 70% of the APIs). For example, is the *mbid* a String or an Integer, can the artist name contain only one word or can it be a concatenation of words. In the case of an optional parameter value, how are the results computed if no input is provided (*lang* is not required, so which value is taken if the user does not provide an input). There is also no information on how the format of the output can be specified (are there any other options, besides XML?). In general, underspecification of interfaces described in HTML is very common [MPD10a] and the Last.fm example clearly demonstrates that. This situation is aggravated by the fact that the Last.fm interface is one of the best documented and structured APIs.

[1] `http://www.last.fm/api`

Furthermore, as seen in Chapter 6, only some APIs are invocable directly via a parameterised URI, while other APIs require the custom construction of the complete HTTP request, specifying in detail the HTTP body. In addition, some APIs are resource-based (i.e. RESTful services), while others are defined in terms of operations, that sometimes even contradict the semantics of the HTTP method used, as in the cased with *hybrid* APIs (for example, calling a getNews operation via HTTP PUT).

In summary some of the most common challenges related to invocation are:

1. Missing information that needs to be compensated by making assumptions or by trial and error;
2. Different ways of transmitting the input;
3. Interweaving of data-payload and invocation relevant details (for example, specifying the format of the input together with how it is transmitted to the server).

Some of these challenges have already been addressed as part of existing invocation solutions, which were discussed in more detail in Chapter 4 and which are analysed in terms of the invocation support that they provide in Section 8.3.

8.3 Requirements

Since currently the majority of the Web API documentation is provided in a human-oriented form as webpages, API users have to invest a lot of manual effort into finding services, interpreting their descriptions and developing hard-wired implementations. The importance of invocation support has already been recognised by some API providers, who deliver custom client libraries in order to ease the use of individual APIs or a particular type of API, such as strictly RESTful ones. However, even with these, further implementation work would still be required and the resulting clients support only individual APIs. This situation is aggravated by the fact that, as already mentioned, Web APIs are characterised by heterogeneity and underspecification of documentation [MPD10a]. The here proposed approach for supporting the invocation of Web APIs builds on existing HTML documentation, where service properties are marked and enhanced with semantic annotations.

In this section we focus on deriving the requirements (marked with '**R**') for a semantic model, capable of supporting the automated invocation, so that given a description, the API can directly be invoked by our invocation engine, without further implementation efforts or completion of manual tasks. In particular, this includes the automated generation of the HTTP request message,

encompassing the input data transformations, creation of the invocation URL and composition of HTTP body, the sending of the request and the subsequent processing of the response message.

In the context of developing invocation support, we aim to cover the majority of the Web APIs, thus providing a general solution and widely applicable support. In practice, there is nothing that makes the Web API invocation a particularly challenging task. In fact, implementation-wise Web API invocation is equivalent to sending an HTTP request and processing an HTTP response. Therefore, no matter what the underlying technology is, manual invocation by the user or via an implemented solution, the invocation of an API comprises the following three steps:

1. ***Construct HTTP request*** – Identify the HTTP Method, construct the invocation URI, prepare the input data, construct the HTTP body and header, craft the HTTP request message.
2. ***Actual invocation*** – Send the prepared HTTP message and receive the Web API server response.
3. ***Process the HTTP response*** – Response handling (parse the response data, response codes, headers, body), process the output data, present the output, error handling.

FIGURE 8.2: Invoking a Web API

As visualised in Figure 8.2 the focus is on determining the information relevant for supporting the automation of three main tasks. Following is a detailed analysis of the resulting requirements, based on a decomposition of the identified steps, starting with preparing the HTTP request.

- **R1: The HTTP method should be explicitly specified as part of the API description**. One of the key elements that determine how the HTTP request is sent is the HTTP method. However, as detailed in Chapter 6, we found out that currently about 40% of the APIs do not state the HTTP method to be used [MPD10a]. This is possibly because providers assume that the method to use is GET, especially for APIs that can be invoked directly through parameterising the URI. Still, this means that the existing API documentation needs to be interpreted by the developer and the correct functioning of the communication

with the server needs to be determined on a trial-and-error basis. Existing approaches for semantically describing APIs, such as MicroWSMO and SA-REST already include the HTTP method as an annotation element. Similarly, XML-based interface description languages such as WSDL and WADL also specify the HTTP method.

- **R2: The description model should support parameterised URIs**. The lack of a unified approach for describing APIs directly influences the way of constructing the invocation URI because some APIs require composing a URI by filling in specific parameter values, while others specify the input as part of the HTTP body. The situation is aggravated by the fact that sometimes key information, such as the input datatypes[2], is missing. This requirement is directly deduced from the observation that currently a big portion of the APIs is based on invocation over a parameterised URI (almost 80%, see Chapter 6).

- **R3: The description model should support the definition of an invocation address**. Web APIs cannot be called, without knowing where the endpoint is available. Therefore, specifying where the Web API can be invoked is crucial.

The requirements regarding the HTTP method (R1) and the Web API endpoint (R3) show some overlap with the ones defined while developing MSM. The reasons for this are twofold. First, we aim to develop a description model that captures all the Web API characteristics that are relevant for invocation. Second, we want to enable the reusability of the model, independently of whether it is used in conjunction with MSM or with any other service conceptualisation.

- **R4: The relationship between the input parts and the HTTP requests should be specified**. An important part of creating the HTTP request is constructing the HTTP body and header. Even though, many APIs transmit the input data directly as part of the invocation URI, it is also very common, especially in the cases were new resources are created or published, that the data payload is sent as part of the HTTP body. Similarly, some input values, such as authentication credentials, are transmitted directly as part of the HTTP header. In fact, about one third of the APIs require the construction of the complete HTTP request, while the rest can be called only by passing input directly via the URI (see Chapter 6). Therefore, it is necessary to be able to specify the parts of the HTTP requests that are used to transmit the input.

- **R5: The input data transformations should be defined**. The preparation of the input data is one of the most challenging tasks related to invocation. Providers commonly use parameters with optional values, default values, alternative values (for example: 1, 2 or 3) and coded values ('en' instead of 'English'). In addition, more than two thirds of

[2]Datatypes are relevant in order to be able to determine the correct format of the input data. It is relevant if the input is, for example, an integer, a range, e.g., 0-1, or a decimal. Similarly, for example, with dates, it is relevant whether only the month or the month and the year are taken as input. The correct formatting and structuring of the input is important for providing invocation support.

the APIs do not explicitly state the datatype of the input, leaving it to the developer to experiment and guess the correct one. In order to avoid having to process data on the syntactic level, but more importantly, to be able to benefit from the automation support provided by describing the APIs and their inputs and outputs semantically, we handle the preparation of the input data with the help of lowering schema mappings. The lowering schema (see Chapter 3) defines how to transform the semantic input into the parameters used in the invocation HTTP message. This approach, which is also adopted by both SA-REST and MicroWSMO, enables the handling of input and output data based on its semantics, not having to deal with the particular format and structure, such as XML or JSON. Therefore, the definition of input data transformations, in terms of lowering schemas, is necessary.

- **R6: Web API characteristics that directly determine how invocation is done should be captured**. More than half of the APIs use optional parameters, which means that the invocation can be completed even if no input values for these parameters are provided. Similarly, despite the fact that according to the REST principles, the output format should be determined via content negotiation, a big number of the APIs do this via an input parameter (for example, http://my.aip.com/getInformation?format=xml). Therefore, the capturing of characteristics that have a direct impact on invocation, such as optional values or output-format parameters, is essential.

- **R7: The description model should support capturing the input as a whole, as well as its individual parts**. The previous requirement leads to the need to be able to describe not only the input and output as a whole but also parts of the message individually. SA-REST introduces the `Parameter` class for this purpose, while MicroWSMO supports only annotations directly on input and output message level. However, it is necessary to support the description of the complete messages as well as the explicit annotation of their message parts, especially because in the context of invocation, some parts of the input modify the actual process of invocation, while others are simply part of the transmitted data payload.

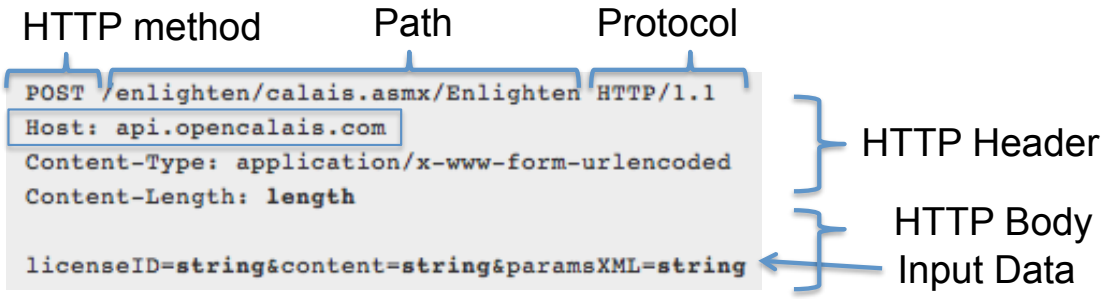


FIGURE 8.3: Composed HTTP Request

Figure 8.3 shows how the crafted HTTP request can look like, with all the necessary input, including the HTTP method, the invocation URI (consisting of the used protocol, the host and the actual path), the HTTP header and body. The *content-length* would have an actual value, depending on the transmitted strings. The so constructed request message can then be sent to the API server.

Once the HTTP method, the invocation URI, the HTTP body and header, and the input data are prepared, the actual HTTP message can be constructed (Figure 8.3). The Web API invocation itself, including sending the HTTP request and receiving the response, is realised as part of the system implementing the invocation engine (see Section 8.5.1) and does not result in any direct requirements for the Web API description model.

- **R8: The output data transformations should be defined**. Once the HTTP response is received, the actual output, which is sent as part of the body, needs to be extracted. There are a number of possible formats for the output, such as HTML, RSS, and CSV, however, providing support for the use of XML and JSON would cover the majority of the APIs (see Chapter 6). Since we are basing our approach on the use of lightweight semantics, the definition and inclusion of a lifting schema mapping is required.

Figure 8.4 shows a sample response, with the corresponding response code and the output data that needs to be extracted and lifted to RDF.

FIGURE 8.4: HTTP Response Handling

- **R9: The output data transformations should be able to handle custom errors**. If the API invocation runs smoothly, the output is extracted and lifted to RDF that can be presented to the end-user (Figure 8.4), or as an alternative, be used as an input for another API, which is part of a service composition. However, if the invocation fails, an appropriate mechanism for error handling needs to be implemented. Ideally, the API should use the standard HTTP error codes in order to indicate what went wrong. However, as we found out, a big portion of the APIs use custom errors, which are transmitted in the HTTP body, instead of the expected output data, while the client receives an HTTP "200 OK" status code from the server. In that case, the lifting schema mapping needs to be able to process custom errors and transform them to RDF that can be presented to the user or the client application.

In summary, we have identified the main steps for completing the invocation process and have derived the resulting requirements. As visualised in Table 8.1, currently none of the existing description approaches cover all of the necessary details. The percentages are based on the results of the first Web API survey (see Chapter 6 for more details). WSDL and WADL, in combination with the support for adding metadata provided by SAWSDL, could be one possible solution. However, as clearly demonstrate by the results of the two Web API surveys, neither WSDL nor WADL is widely adopted. In fact, none of the APIs that we analysed had a documentation based on one of the description forms.

The two main gaps that we have identified are in *providing means to specify the data grounding*, i.e. which parts of the input are transmitted via which parts of the HTTP request (in the URI, in the HTTP body or in the HTTP header), and *enabling the description of individual input parts*. In particular, currently it is not possible to differentiate between input that is simply transmitted as part of the request and input that actually influences the way, in which the invocation is performed, such as output format or authentication credentials parameters.

TABLE 8.1: Requirements Coverage

Description	R1: HTTP Method	R2: Param. URI	R3: Invoc. URI	R4: Input Address	R5: Lo Mapping
HTML Doc.	60.4%	77.3%[1]	96.4%	100%	N/A
WADL	Yes	Yes	Yes	Yes	via SAWSDL
WSDL 2.0	Yes	Yes	Yes	Yes	via SAWSDL
MicroWSMO	Yes	Yes	Yes	No	Yes
SA-REST	Yes	Yes	Yes	No	Yes

	R6: Opt. Param & Further	R7: Msg. Parts	R8: Li Mapping	R9: Custom Errors
HTML Doc.	61%-opt. param	N/A	N/A	∼50%
WADL	Yes	Yes	No	No[2]
WSDL 2.0	Yes	Yes	No	No[2]
MicroWSMO	No	No	Yes	Yes
SA-REST	No	No	Yes	No

[1] Percentage of APIs where the URI uses query parameters, numbers from the second Web API survey;

[2] No support for custom errors.

Therefore, existing lightweight semantic approaches enable the creation of very basic annotations that lack some details that are key for supporting invocation, such as data grounding, parameterised URIs and output format selection. Our goal is to provide a common basis for overcoming the current API heterogeneity by devising a unified invocation solution, which also encourages less underspecification by defining concrete guidelines for the details that an API description needs to include. In the following section we design a model, satisfying the description requirements derived in this section.

8.4 Web API Grounding Model

In this section we introduce the Web API Grounding Model. In particular, we explain the design decisions that we have made and focus on describing the main parts of the model, which were especially introduced in order to support Web API invocation. It is important to point out that in this chapter we focus solely on service properties required for constructing the HTTP request and processing the HTTP response, assuming that authentication is handled separately. We handle the modelling of commonly used authentication approaches in the following chapter, advocating a solution that is based on modularity and extensibility. Since we follow an approach based on enhancing and reusing previous work, we implement the invocation support by extending the Minimal Service Model, described in the previous chapter.

8.4.1 Design Decisions

Based on the results of the analysis conducted in the previous section and taking into consideration existing service description approaches, we aggregate the collected data and devise an invocation model, which enables the creation of semantic Web API annotations capable of supporting the automated invocation of the majority of the APIs. The process of defining the model was guided by a number of design decisions. In particular, as discussed in the previous chapter, we follow the already described ontology engineering approach and, furthermore, advocate clarity, coherence, extensibility, minimal encoding bias and minimal ontological commitment.

Furthermore, the development process was lead by a number of competency questions used to determine the concepts and their relationships, which are to be captured by the model. We started by identifying the main service properties that are important in the context of invocation. Relevant information in this respect is: *"What are the main elements required for invocation?"*, *"Are there relationships between them?"*. These questions can be refined: *"What are the addresses for service X?"*, *"What are the addresses for operation X?"*, *"What are the relationships between a service address and an operation address?"*, *"What are the HTTP methods of operation X?"*. In terms of the input details we can specify *"What are the different parts of input X?"*, *"What are the relationships between the input parts?"*, *"How is the input data transformation for input/input part X done?"*, *"What is the content type of input/input part X?"*, and similarly for the output – *"What are the different parts of output X?"*, *"What are the relationships between the output parts?"*, *"How is the output data transformation for output/output part X done?"*, *"What is the content type of output/output part X?"*. Further details related to the input include *"Which part of the HTTP message is used to send input/input part X?"*. We also capture error details in terms of *"What are the errors that can occur before the calling of operation X?"*, *"What are the errors that can occur after the calling of operation X?"* In addition to

the competency questions, used for identifying the information that needs to be captured by the Web API Grounding Model, we adopted some complementary design decision.

In particular, we take an operation-based view on the interface definition, as opposed to a resource-based one. Even though, RESTful services and Web APIs are often used as synonyms, actually only about a third of the APIs have RESTful interfaces [MPD10a]. Since we do not have access to the actual implementation of the APIs, we are not able to determine what portion of these APIs is truly RESTful. Therefore, the percentage is most probably lower, or equal at best. As already pointed out, currently APIs can be classified into three different types: RESTful, RPC-style and Hybrid [RR07] and despite the fact that the proliferation of Web APIs is frequently attributed to the application of the REST paradigm [Fie00], the majority of the APIs (67%) are described in terms of operations. Therefore, in order to ensure wide coverage[3], the developed description model is based on defining the API in terms of service and operation elements (as opposed to resources).

Another design decision is related to the way of describing the address. In particular, we have observed that it is common for one API to have a number of operations, that share the same domain as part of the invocation URI. For example, the Last.fm operations *artist.getInfo* `http://ws.audioscrobbler.com/2.0/?method=artist.getinfo` and *album.getInfo* `http://ws.audioscrobbler.com/2.0/?method=album.getinfo` share the same service URL `http://ws.audioscrobbler.com/2.0/`. Therefore, we should assign an address to the service, so that it can be overwritten or further specialised by the definition of individual operation addresses. This idea is exemplified in Listing 8.1, where the common Last.fm part of the URI is further specialised by adding the corresponding operation address part.

```
1 Last.fm service address = http://ws.audioscrobbler.com/2.0/
2 −artist.getInfo  operation address = ?method=artist.getinfo&artist=Cher&api_key=xxx
3 −album.getInfo operation address = ?method=album.getinfo&api_key=xxx
```

LISTING 8.1: Service and Operation Definition

Finally, we also take into consideration design decisions that support the reusability of the developed solution. For instance, the model should be extensible to cover further characteristics. Furthermore, the model should be easy to interpret and use while generating annotations and, finally, the conceptualisation of the invocation relevant information should be designed independently of the implementation solution in the form of an invocation engine.

The competency questions and the identified requirements (Section 8.3) ensure that the resulting ontology can capture the service properties and their relationships, which are especially relevant for supporting invocation. In the next section we describe the Web API Grounding Model in detail.

[3]Similarly to MSM, in the context of the Web API Grounding Model, we define "wide coverage" as being able to describe at least 80% of the Web APIs.

8.4.2 Extending MSM with Invocation Support

The Minimal Service Model (MSM) aims to capture the core semantics of services, thus being able to provide basic support for handling Web APIs and traditional Web services in a unified way. However, MSM is not particularly designed to support invocation and in its first version, this task was only approached in a simplistic manner, covering only APIs that are directly invocable over the URI, with matching parameters, and input explicitly via the parameter names. In this section, we describe how we have extended MSM in order to support the automated invocation of the majority of the Web APIs.

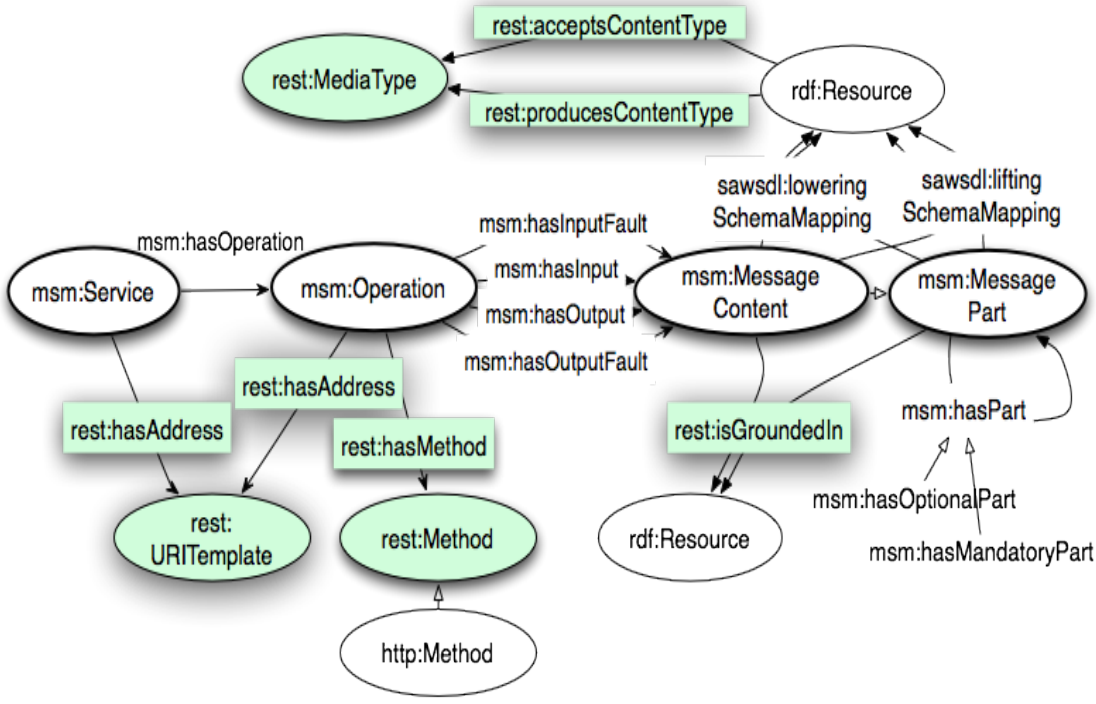


FIGURE 8.5: Web API Grounding Model

The Web API Grounding Model is kept very simple and the main extensions are realised as part of the *rest:* namespace, as visualised in Figure 8.5. We have designed the model by providing a strict decoupling between the parts that capture the core service properties (shown in the figure within the *msm* namespace) and the specific extensions for supporting invocation (within the *rest:* namespace). Figure 8.5 provides a general overview of the Web API Grounding Model, consisting of the *msm* namespace elements[4], including *Service*, *Operation*, *MessageContent* and optional or mandatory *MessagePart* (see Chapter 7 for further details on MSM), and the *rest* namespace elements[5] that represent the main extensions to the model in order to support invocation. In particular, we have included three main extensions in order to better facilitate invocation:

[4] `http://purl.org/msm`
[5] `http://purl.org/hRESTS/`

1. Definition of *isGroundedIn* property for specifying how the input parameters are transmitted.

2. Definition of the service and operation addresses in terms of *URI templates*, in order to enable the grounding of inputs as part of the URI but also to facilitate the definition of a common service URI and its specialisation through individual operation URIs.

3. Definition of *acceptsContentType* and *producesContentType* properties for specifying the input and output formats of the service.

isGroundedIn. The *isGroundedIn* property is used to define, in which part of the HTTP request the input is transmitted. We differentiate between three options for specifying the input grounding – as part of the URI, in the HTTP body or as an HTTP header parameter. Therefore, *msm:MessageContent* and *msm:MessagePart* can have *rest:isGroundedIn* linking to a rdfs:Resource, where the grounding of a parameter is identified through the HTTP vocabulary [Con11] (*http:Body* or an instance of *http:HeaderName*) or a literal String (name of a parameter in URI template). The need to transmit parameters via the HTTP header occurs only very rarely, such as in cases where the authentication credentials are defined and sent as part of the input. We include this option for data grounding in order to ensure completeness. In contrast, sending input through parameter values in the URI is very common and requires linking the address URI template values to the particular message content or message part definition. This approach is usually used in APIs that enable the querying and retrieval of data. On the other hand, posting content or sending a complex object such as XML or JSON is commonly done by sending it in the HTTP body.

Listings 8.2 and 8.3 provide two API operation descriptions that exemplify how input can be passed in the URI (Listing 8.2, see Lines 9, 13, and 15) or in the HTTP body (Listing 8.3, see Lines 9 and 11). In summary, the HTTP vocabulary allows the *isGroundedIn* property to identify that the result of the lowering transformation of the input becomes the value of a particular HTTP header, or of the HTTP body. If the *isGroundedIn* property has a literal value, it names the URI parameter that will contain the value of the lowering transformation.

```
1  @prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
2  @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
4  @prefix msm: <http://purl.org/msm#>.
5  @prefix rest: <http://purl.org/hRESTS#>.
6
7  :ArtistGetInfo  a msm:Operation;
8     msm:hasInput :ArtistGetInfoInput;
9     rest:hasAddress "method=artist.getinfo&artist={p1} &api_key={p2}"^^rest:URITemplate.
10 :ArtistGetInfoInput  a msm:MessageContent;
11    msm:hasPart :artist, :api_key.
12 :artist  a msm:MessagePart;
13    rest:isGroundedIn "p1"^^rdf:PlainLiteral.
14 :api_key a msm:MessagePart;
15    rest:isGroundedIn "p2"^^rdf:PlainLiteral.
```

LISTING 8.2: Example of Input Data Grounding I

```
@prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix http: <http://www.w3.org/2008/http−methods#>.
@prefix msm: <http://purl.org/msm#>.
@prefix rest: <http://purl.org/hRESTS#>.

:ArtistGetInfo  a msm:Operation;
   msm:hasInput :ArtistGetInfoInput;
   rest:hasAddress "method=artist.getinfo"^^rest:URITemplate.
:ArtistGetInfoInput  a msm:MessageContent;
   rest:isGroundedIn http:body.
```

LISTING 8.3: Example of Input Data Grounding II

Similarly to the approaches presented in the previous chapter, related to employing MSM for creating semantic Web API descriptions, the Web API Grounding Model can either be used directly to create a description (such as the one visualised in Listing 8.2) or it can be used in a bottom-up manner. In this case, existing HTML documentation is structured with the help of tags that represent the grounding model via a mapping on the syntactic level. Similarly to MSM, here again we use hRESTS tags, based on the grounding model.

Given the webpage describing an API, the input grounding needs to be defined already on the level of HTML markup, which can then subsequently be extracted to RDF. Therefore, we also define some extensions to the hRESTS elements inserted within the HTML documentation. Listing 8.4 shows how the description of input that is transmitted via the "loc" URI parameter and input that is sent as part of the HTTP body looks like. A complete list of the hRESTS extensions for the Web API Grounding Model is given in the Appendix in Section A.2.

```
Input in  URI parameter: <span id="grounding1" class="grounding" title="loc"/>
Input in  HTTP body: <span id="grounding2" rel="grounding" href="http://www.w3c.org/2006/http#body"/>
```

LISTING 8.4: Input Data Grounding through hRESTS

URI Templates. The definition of the service address in terms of URI templates was already introduced in the first version of hRESTS. We build on this solution in order to enable the specification not only of the service address but also the operation address with the help of URI templates. Moreover, we enable the extension and overwriting of the service address through the specification of individual operation addresses. In this way we not only support the definition of a common API domain address for the service but also directly support the grounding of the input data by enabling the linking of URI template individual parameters to separate input content parts. Listing 8.5 visualises how the usage of URI templates is applied in defining the API addresses.

The usage of URI templates for defining the service and operation addresses does not require the definition of new hRESTS elements used to markup the HTML documentation. This is done through the usage of the *address* class (`<span id="address1" class="address">`

`http://unl.myurl </span>`). Still it is up to the available user support in terms of annotation tools, such as SWEET (see Chapter 10.2), to ease the creation of the URI template, for example, by automatically adding the defined input parameters.

```
1  @prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
2  @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
4  @prefix msm: <http://purl.org/msm#>.
5  @prefix rest: <http://purl.org/hRESTS#>.
6
7  :LastFMService a msm:Service;
8     msm:hasOperation :ArtistGetInfo;
9     rest:hasAddress "http://ws.audioscrobbler.com/2.0/?"^^rest:URITemplate.
10 :ArtistGetInfo  a msm:Operation;
11    rest:hasMethod "GET";
12    rest:hasAddress "method=artist.getinfo&artist={p1}&api_key={p2}"^^rest:URITemplate.
```

LISTING 8.5: Using URI Templates

acceptsContentType and producesContentType. These two properties were introduced in order to be able to specify the input and the output of a service. Given the input RDF data, the lowering schema mapping must lower the RDF data to a format supported by the actual Web API, for example JSON or XML. Similarly, the response message body needs to be lifted to RDF using a lifting schema mapping that supports its actual format. In other words, this information is required in order to be able to select a lifting schema mapping that supports the content type of the particular response message. Therefore, additional metadata is provided on lowering and lifting schema mappings:

- For lowering mapping, the content type that the mapping produces (using the property *producesContentType*) is described;
- For lifting mapping, the content type that it can process (*acceptsContentType*) is described.

Both properties point to a literal String that names a MIME media type. Listing 8.6 exemplifies how the type resulting from the input transformation and the type required for the output transformation can be specified. In this case both types are defined as *application/xml*.

```
1  :ArtistGetInfoInput  a msm:MessageContent;
2     msm:hasPart :part1.
3  :part1 a msm:MessagePart ;
4     rest:isGroundedIn http:body ;
5     sawsdl:loweringSchemaMapping  "http://example.com/lowering.xsparql" .
6  "http://example.com/lowering.xsparql" rest:producesContentType "application/xml"^^rest:MediaType .
7
8  :ArtistGetInfoOutput a msm:MessageContent;
9     sawsdl:liftingSchemaMapping "http://example.com/lifting.xsparql"  .
10 "http://example.com/lifting.xsparql"     rest:acceptsContentType  "application/xml"^^rest:MediaType .
```

LISTING 8.6: Example for Produces and Accepts Content Type

In addition, since lifting and lowering schema definitions can be assigned to both the message content, as well as its individual parts, the accepted and produced content types can be used on the same level of granularity. Therefore, we can define the input and output formatting on the same level (e.g. message content or message part) as the transformation schema definition. We also provide hRESTS extensions for including the corresponding markup within the HTML documentation. Listing 8.7 provides an example for specifying the expected output format of the lowering transformation, so that the input is accepted by the API and the invocation can take place.

```
1 <span class="parameter" rel="lowering"  href="lowering.xsparql">
2     <span  rel="grounding"  href="http://www.w3c.org/2006/http#body"  />
3     <span  class="content-type" title="application/xml"  />
4 </span>
```

LISTING 8.7: hRESTS Extensions for Produces and Accepts Content Type

The here listed three main extensions to MSM represent the core enhancements for enabling Web API invocation. However, we have also implemented a number of auxiliary annotation options that aim to capture further invocation-relevant details. In particular, we differentiate between input parameters that represent actual service data payload and parameters that determine how the API is called (specified in R7). Two main types of input parameters that pose restrictions on the way the HTTP request is crafted are authentication credentials (discussed in detail in the following chapter) and output format parameters (input parameters that are used to specify the format of the output). Therefore, we provide hRESTS extensions for capturing output format parameters within the HTML documentation, with the possibility to also extract them as part of the resulting semantic Web API description.

```
1 <span class="parameter-output-format">param1</span>
```

LISTING 8.8: hRESTS Extensions for Output Format

Listing 8.8 shows the new class for describing the input parameters, while Listing 8.9 visualises the resulting RDF.

```
1 :param1 a msm:MessagePart, rest:OutputFormatParameter .
```

LISTING 8.9: Resulting RDF for Output Format

Restrictions on Message Parts and Grounding Definitions. With the introduction of the *MessageContent* and *MessagePart* classes in MSM (Chapter 7) the description of the API's inputs and outputs becomes much more flexible. However, this flexibility brings a certain level of complexity when it comes to processing the individual grounding definitions and lifting and lowering transformations, in the context of invocation. In order to better facilitate their processing but also to ensure that the descriptions are complete and consistent, we define the following restrictions:

- **Multiple lowering (lifting) schema definitions** can be defined for one *MessagePart* or *MessageContent* instance. Multiple lowerings (liftings) per *MessagePart*/*MessageContent* are used for providing alternatives (for example, for supporting different output formats).
- **At least one lowering transformation** should be defined per *isGroundedIn* on *MessagePart*/*MessageContent*. No lowering is allowed on *MessagePart*/*MessageContent* without *isGroundedIn* element assigned to it.
- Grounding and lowering (lifting) is allowed either on *MessageContent* or on all its direct *MessageParts*. This is necessary in order to avoid situations, where overlapping groundings or transformations (both on the level of message content and parts) are defined and it is not clear, which ones should be used.

These restrictions are not necessary for using the Web API Grounding Model for creating semantic Web API descriptions that support automated invocation but rather ensure that the processing by the invocation engine, introduced in the next section, goes smoothly. In particular, they are meant to ensure that all data transformations are properly defined and that all grounded inputs can also be lowered.

```
1  @prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
2  @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
3  @prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
4  @prefix msm: <http://purl.org/msm#>.
5  @prefix rest: <http://purl.org/hRESTS#>.
6  :LastFMService a msm:Service;
7     msm:hasOperation :ArtistGetInfo;
8     rest:hasAddress "http://ws.audioscrobbler.com/2.0/?"^^rest:URITemplate.
9  :ArtistGetInfo a msm:Operation;
10    msm:hasInput :ArtistGetInfoInput;
11    msm:hasOutput :ArtistGetInfoOutput;
12    rest:hasMethod "GET";
13    rest:hasAddress "method=artist.getinfo&artist={p1}&api_key={p2}"^^rest:URITemplate.
14 :ArtistGetInfoInput a msm:MessageContent;
15    msm:hasPart :artist, :api_key.
16 :artist  a msm:MessagePart;
17    sawsdl:loweringSchemaMapping "http://iserve.kmi.open.ac.uk/lilo/ArtistLowering.txt";
18    sawsdl:modelReference "http://purl.org/ontology/mo/MusicArtist";
19    rest:isGroundedIn "p1"^^rdf:PlainLiteral.
20 :api_key a msm:MessagePart;
21    sawsdl:loweringSchemaMapping "http://iserve.kmi.open.ac.uk/lilo/APIKeyLowering.txt";
22    sawsdl:modelReference  "http://purl.oclc.org/NET/WebApiAuthentication#API_Key";
23    rest:isGroundedIn "p2"^^rdf:PlainLiteral.
24 :ArtistGetInfoOutput a msm:MessageContent;
25    sawsdl:liftingSchemaMapping "http://iserve.kmi.open.ac.uk/lilo/ArtistGetInfoLifting.txt".
```

LISTING 8.10: Example RDF Service Description

Listing 8.10 shows the semantic description of the Last.fm[6] *artist.getInfo* operation, which is created based on its HTML documentation. The description is created with the help of the annotation tool SWEET (see Chapter 10.2) [MKP09] that supports the creation of semantic Web API descriptions. The *LastFMService* contains an *ArtistGetInfo* operation with input

[6] `http://www.last.fm/api/show?service=267`

ArtistGetInfoInput. The input contains message parts *artist*, *api_key*, and more importantly their links to external ontology entities, i.e. to Music Ontology, as well as the links to loweringSchemaMapping and liftingSchemaMapping scripts, which are in the form of XSPARQL queries [AKKP08][7]. The input data grounding is realised through the definition of the operation address as a parameterised URI Template, where each of the message parts has an *isGroundedIn* property specifying its place in the URI.

In summary, the main enhancements that we have introduced in order to enable automated Web API invocation through the Web API Grounding Model are support for data grounding, URI template definition and overloading, and fine-grain and type-sensitive data transformations. The model is defined in a modular way so that the extensions realised as part of the *rest:* namespace and the core *msm:* can be used independently or in combination with further service description models. For example, the underlying service model can be exchanged and the hRESTS extensions can be used on their own. In addition, WSMO-Lite can be used on top of the Web API Grounding Model to specify functional and non-functional semantics, as well as conditions and effects.

8.5 Implementation

In this section we show how semantic Web API descriptions based on MSM and the Web API Grounding Model can be used to support the automated invocation of services. In particular, we present a design and implementation of an invocation engine – OmniVoke [LPK+11], that given a semantic Web API description, completes the invocation process automatically.

8.5.1 OmniVoke

We show the practical applicability of the developed Web API Grounding Model by demonstrating how Web API descriptions, expressed in its terms, can automatically be invoked through an actual invocation engine. In particular, the contribution described here represents a general invocation engine – OmniVoke [LPK+11], that provides a "meta" API that serves as common invocation point for the majority of the APIs. OmniVoke is not one of the contributions of this thesis but is rather used as a proof-of-concept implementation. The invocation engine is exposed as a RESTful API, which can be called to invoke a particular Web API, given its semantic description and the data payload in RDF. Therefore, OmniVoke turns every API into an RDF consuming and producing service. This builds the foundations for enabling the direct integration with Linked Data, where the input can be a LD source and the output can be fed into the LD

[7] XSPARQL, see `http://xsparql.deri.org`

cloud as well, thus contributing directly towards achieving the vision of Open Services on the Web.

The main goal of OmniVoke is to provide an entry point for the invocation of Web APIs through their semantic descriptions, thus enabling the automation of this service task. Furthermore, since the API inputs and outputs are semantically described, and with data sources on the Web undergoing a developing trend towards Linked Data [BHBL09], the invocation engine is a step in the direction of supporting the integration of APIs with Linked Data.

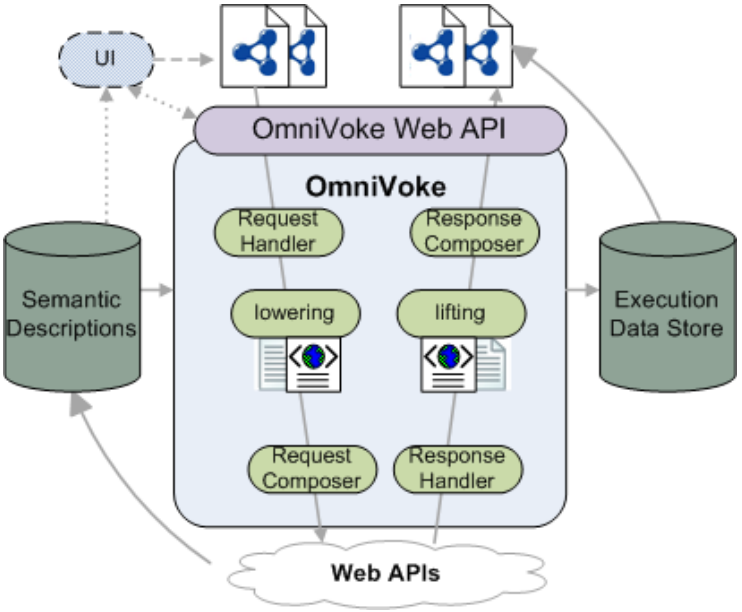


FIGURE 8.6: OmniVoke Architecture

Figure 8.6 visualises OmniVoke's architecture, which is aligned with the process of completing an HTTP request and processing an HTTP response (as described in Section 8.3). OmniVoke comprises a number of components, including the *Request Handler*, which is triggered when an invocation request is received and carries out the tasks of validating and de-capsulating the invocation request. Initially the service description is retrieved via the service UID. After that the *Lowering* component undertakes the task of transforming the RDF input data to the format supported by the actual API. It works by executing the lowering scripts designed for each input that requires lowering. If there are more than one transformation definitions, the correct script is matched to the correct format via the *acceptsContentType* property values. XSLT[8] together with SPARQL[9] have been used widely within the community [Bis12]. Lately, XSPARQL [AKKP08], which combines XQuery[10] and SPARQL, has been recognised as a more effective language due to its advantages of being able to load the RDF and use SPARQL to retrieve the values

[8] `http://www.w3.org/TR/xslt/`
[9] `http://www.w3.org/TR/rdf-sparql-query/`
[10] `http://www.w3.org/TR/xquery/`

needed, thus avoiding having to create XSLT scripts that can deal with every possible RDF serialisation [AKKP08].

After the input data is prepared and using information given in the service description, a valid HTTP request for invoking the actual Web API can be constructed by the *Request Composer*. The created HTTP message is subsequently sent to the Web API server. Once the response message is returned, the *Response Handler* is triggered to extract the output information (mainly status code and response data) out of the response header and body, and to decide whether lifting is required for each output, with the help of the information in the service description. The *Lifting* component carries out the execution of the data transformation scripts attached to the output that requires lifting, as annotated in the service description. Similarly to lowerings, lifting scripts can be written in XSPARQL. Finally, after the output is lifted to RDF, a new response, comprising only RDF data, is constructed by the *Response Composer* and returned to the client as the final response to the initial invocation request.

As can be seen, the internal design of OmniVoke is very much aligned with the process of completing and actual Web API invocation. The main enhancements are that it takes RDF data as input and returns RDF data as output and is capable of directly completing all invocation steps only based on the details available in the semantic service description and without any additional implementation or manual work. Therefore, OmniVoke allows seeing any Web API as RDF consumer and producer, thus supporting the direct integration of Web APIs and Linked Data. For instance, the Web API input can be provided with the help of SPARQL queries ran against an existing data repository or alternatively, an appropriate user-interaction interface can be used to allow the user to specify the input. As a result, OmniVoke contributes towards improving the overall integration potential of APIs, in the context of creating mashups and building applications, since Linked Data has proven to be an adequate technology for sharing and integrating data in distributed settings.

Every invocation request to a Web API is raised to OmniVoke through a homogeneous interface, via a URI that contains the identity of the API semantic description. The invocation of individual operations of services is done by passing the ID of the semantic Web API description, based on the Web API Grounding Model, and the name of the particular operation, which is to be called. Given the invocation URI, the request data is sent to the invocation engine in the HTTP body via POST, since the calling of the invocation API represents the creation of a new request resource.

Given the Last.fm description in Listing 8.10, the API can directly be invoked by OmniVoke. Listing 8.11 shows a sample RDF input that can be posted in the HTTP body, in order to call the Last.fm *ArtistGetInfo* operation.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
  xmlns:waa="http://purl.oclc.org/NET/WebApiAuthentication#"
  xmlns:mo="http://purl.org/ontology/mo/"
  xmlns:foaf="http://xmlns.com/foaf/0.1"
  xmlns:sioc="http://rdfs.org/sioc/ns#">
    <mo:MusicArtist rdf:about="#artist1">
      <foaf:name>Cher</foaf:name>
    </mo:MusicArtist>
    <sioc:UserAccount rdf:about="#usr0">
      <waa:API_Key>b25b959554ed76058ac220b7b2e0a026
      </waa:API_Key>
    </sioc:UserAccount>
</rdf:RDF>
```

LISTING 8.11: Example RDF Input Data

This RDF input is then lowered to the actual format of the input expected by the API, with the help of XSPARQL transformations given in Listing 8.12.

```
declare namespace waa = "http://purl.oclc.org/NET/WebApiAuthentication#";
declare namespace sioc = "http://rdfs.org/sioc/ns#";
{ for  $apikey $user from <file:StaticInputFile>
      where { $user a sioc:UserAccount;
                    waa:API_Key $apikey.}
 return  {$apikey}}

declare namespace foaf = "http://xmlns.com/foaf/0.1";
declare namespace mo = "http://purl.org/ontology/mo/";
{ for  $artist_name $artist  from <file:StaticInputFile>
      where {  $artist   a mo:MusicArtist;
                         foaf:name $artist_name.}
 return  {$artist_name}}
```

LISTING 8.12: Input Lowering Transformation

When the invocation is completed and the HTTP response message is processed, the actual output of the API is lifted to RDF by using the transformation defined in Listing 8.13.

```
declare namespace foaf="http://xmlns.com/foaf/0.1";
declare namespace mo="http://purl.org/ontology/mo/";
let  $doc :=doc("OriginalOutputFile")
 for  $listing   in  $doc//artist
     let  $name := $listing/name
     let  $id  :=  $listing/mbid
     let  $url  :=  $listing/url
     let  $image := $listing/image[@size='medium']

     construct
          {  _:p a mo:Artist;
                foaf:name  {data($name)};
                mo:musicbrainz_guid  {data($id)};
                mo:homepage {data($url)};
                mo:image {data($image)}; }
```

LISTING 8.13: Output Lifting Transformation

In summary, given the semantic API description and RDF input data, OmniVoke can automatically complete the invocation process without any further implementation work or manual tasks. Therefore, the invocation engine is a way of demonstrating the practical applicability and use of the Web API Grounding Model, in the context of supporting the automated invocation of Web

APIs based on lightweight semantics[11]. By using a unified declarative way of describing APIs and with the support of a general purpose invocation engine, any API is available through a singe interface (that of OmniVoke) and can be directly supplied with the required RDF input, as specified in its semantic description. Therefore, the here presented invocation engine serves as the basis for quickly and dynamically developing Web Applications on top of Web APIs.

For instance, the API descriptions have already been successfully used to create the SOA4All Real Estate Finder iPhone and iPad app, which is a user friendly mobile client for finding real estate. A screencast of the SOA4All Real Estate Finder is available under `http://people.kmi.open.ac.uk/jacek/soa4re-screencast/soa4re-video-black.mp4`. More details about this application and further use-case scenarios, built with the help of the here presented Web API semantic model and implementation technology, are given in Section 11.4.1.

8.6 Summary

This chapter describes the Web API Grounding Model, which serves as the basis for creating semantic Web API descriptions that support the automated completion of the invocation process. We start with a motivating example that clearly demonstrates some of the challenges related to invocation, which result from current common documentation forms. We continue to derive a set of requirements for designing a model capable of supporting invocation, based on a thorough analysis of the individual steps and pieces of data required for completing client-server based communication. Taking into consideration the gathered input, we present the resulting Web API Grounding Model and explain in detail each of the defined extensions, which can be used independently or in conjunction with MSM.

The Web API Grounding Model is realised by carefully analysing and gathering the details that are relevant for supporting automated invocation. This is especially important in the context of determining the support provided by the implementation solution and the coverage that it has. As a result, the model captures all invocation-relevant characteristics and gives the means for creating annotations and semantic Web API descriptions that overcome heterogeneity, thus enabling the handling of APIs in a unified way. In addition, it serves as the basis for counteracting underspecification, since it can be used as a reference point by providers in order to determine, which details need to be included as part of the documentation.

We also demonstrate how the Web API Grounding Model is implemented as part of the OmniVoke invocation engine. In summary, the Web API Grounding Model, in combination with MSM, contributes directly towards achieving the vision of Open Services on the Web.

[11] Evaluation results on tested services are given in the evaluation chapter, Section 11.2.3.

Chapter 9

Automating the Authentication of Web APIs

In this chapter we introduce the extensions to the core service model that were especially derived in order to better support the automation of the Web API authentication task. As already pointed out, we follow an approach based on modularity and extensibility by defining MSM, which is then enhanced with further extensions that are particularly targeted at supporting individual tasks. In the previous chapter we focused on invocation, handling authentication more or less as a black box. In this section we describe our work on authentication in detail. In particular, the defined extensions consist of properties that enable the capturing of credentials, the way that they are transmitted and whether a particular authentication protocol is used. The Web API descriptions, created with the help of the Web API Authentication (WAA) model [MPD+10b], enable the automated completion of the authentication task, as a prerequisite to the invocation process. This is realised with the help of a simple authentication engine, which prompts the user for the necessary credentials and performs the remaining steps of the authentication process, without further user involvement.

9.1 Introduction

As previously emphasised and shown by the study of the current state of APIs on the Web (see Chapter 6), there is a wide range of used authentication approaches, which need to be completed before being able to perform invocation. Authentication is defined as the act of verifying that someone is, in fact, who he/she claims to be. In the context of using Web APIs, authentication involves a number of different mechanisms and types of credentials (see Chapter 4), used for confirming the identity of the user but also focusing on verifying that he/she is allowed to access the service. Therefore, when it comes to Web APIs, the user credentials are simultaneously

used for both authentication and authorisation. That is why in this chapter, when talking about authentication, we also discuss access control.

Currently more than 80% of the Web APIs require some form of authentication (see Chapter 6). Still, up to date, the importance of authentication as part of the invocation process has been overlooked. As our two Web API surveys point out, the majority of the Web APIs require some form of authentication but none of the existing formalisms and annotation approaches deal with this. Moreover, none of the available tools, which provide developer support for creating mashups, such as Yahoo Pipes[1] and DERI Pipes[2], handle authentication in an integrated way and it still needs to be addressed separately. As a result, the invocation of individual Web APIs and their use within mashups, requires additional manual development work, independently of whether the used APIs have semantic descriptions or not. In particular, this involves the implementation of individual clients or API wrappers, that take care of the authentication before the API can be used.

In this chapter we focus on providing a solution towards supporting automated authentication as part of the Web API invocation process. We build on the work on invocation, presented in the previous chapter, and aim to enhance it by providing support for handling authentication details as well. In particular, we propose an extension to MSM, which can be used in conjunction with the Web API Grounding Model, in order to integrate authentication support as part of invocation. We determine the most commonly used authentication approaches by analysing the data collected during the two Web API studies. In the light of these results, we propose an ontology for the semantic annotation of Web API authentication information and show how it can be used to enrich semantic Web API descriptions with authentication details. We also demonstrate the applicability of our approach by providing a prototype implementation, which uses authentication annotations as the basis for automated service invocation, and by covering a range of authentication approaches through exiting Web API examples.

This chapter is structured as follows: Section 9.2 provides a motivating example that illustrates the challenges related to Web API authentication. In Section 9.3 we systematically derive the requirements for designing a model capable of capturing authentication information based on an analysis of common Web API authentication approaches (see Chapter 4). We present the resulting Web API Authentication Model in Section 9.4. Section 9.5 includes a description of the implementation for supporting the automated authentication, by using lightweight semantic annotations, while Section 9.6 completes the chapter by summarising the presented solution.

[1] `http://pipes.yahoo.com/pipes/`
[2] `http://pipes.deri.org/`

9.2 Motivating Example

In this section we reintroduce the example used in the previous chapter in order to demonstrate the necessity of authentication information during the invocation of Web APIs. In particular, we describe one of the operations of the Last.fm API[3].

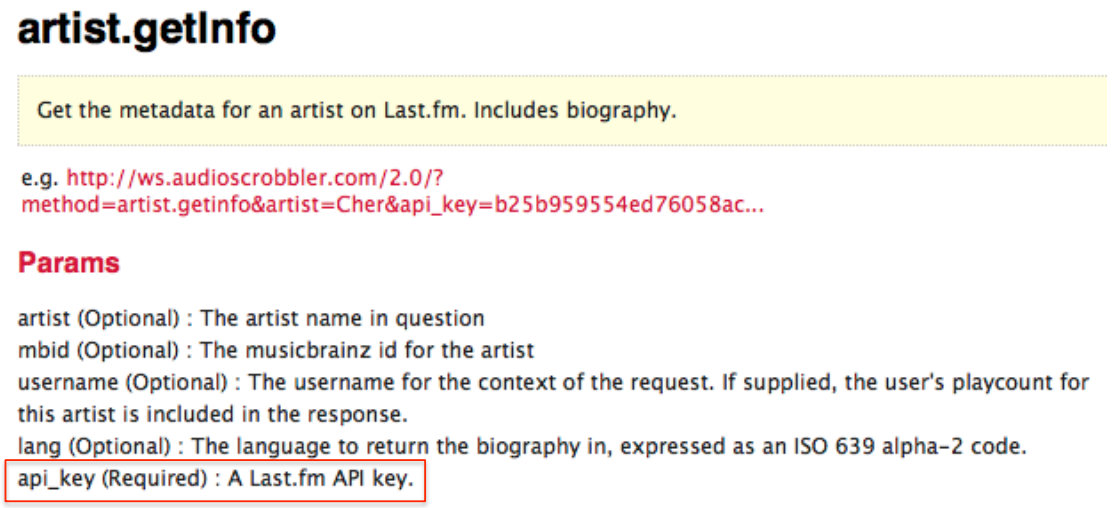


FIGURE 9.1: Extract from the Last.fm API

Figure 9.1 shows the Web API operation for getting the details for a particular artist. The provided data can be used directly or as part of a mashup, where artists' details are combined with latest charts news, for example. As can be seen, the API expects an authentication API key (*api_key* – the last required parameter), without which the API cannot be called.

We analyse common authentication mechanisms and develop an approach for capturing authentication details as part of semantic Web API descriptions. The approach, which we propose differs from WebID, Web-key, XAuth and other authentication mechanisms (see Chapter 4) because we are not suggesting to alter the current Web API authentication landscape by introducing a common standard. Instead, based on a study of current Web API authentication mechanisms, we provide a lightweight model and approach for the annotation of APIs with the corresponding, mechanism-specific, information.

9.3 Requirements

In this section we focus on deriving the requirements (marked with '**R**') for a Web API Authentication Model. In particular, we aim to cover the majority of the Web APIs[4], not narrowing-down

[3] `http://www.last.fm/api`

[4] Similarly to MSM and the Web API Grounding Model, in the context of the Web API Authentication Model, we define "the majority" as being able to describe at least 80% of the Web APIs.

to one particular type of used credentials, such as the API key, or to a particular authentication mechanism. Therefore, the goal is to enhance the previously developed approach for automating the invocation of Web APIs with authentication support, by developing a description model capable of capturing common authentication characteristics.

The overview of common authentication approaches serves as the foundation for identifying three main characteristics for describing each of the approaches: 1) the required credentials, 2) the used authentication protocol, and 3) the way of sending the authentication information. Therefore, we aim to capture these characteristics as part of the authentication model by deriving the following requirements:

- **R1: The used credentials should be specified**.
- **R2: The used authentication protocol should be specified**.
- **R3: The way of sending the authentication information should be specified**.

Furthermore, we use the data collected during the Web API studies, which leads us to the following conclusions regarding authentication:

1. More than 80% of Web APIs require authentication for at least an operation. Therefore, authentication is a vital part of the invocation process and any approach disregarding authentication information has very limited support.
2. Only about 25% of the Web APIs use an authentication approach, which protects the user credentials and/or the content of the message.
3. The currently used authentication approaches are very heterogeneous and there is no common widely accepted way for addressing Web API authentication.

In order to be able to deal with the current heterogeneity:

- **R4: The most commonly used authentication approaches should be covered**. This includes API key-based, HTTP Basic, username and password-based, HTTP Digest and OAuth. Naturally, the goal is to provide a wide coverage, however, if it is not possible to provide support for all authentication approaches, at least the most commonly used ones should be covered. It is important to find a balance in the tradeoff between complexity of the model and the provided coverage. Therefore, we focus on covering the most common approaches and enable the support for additional ones through the definition of extensions.

The same holds for frequently used credentials and ways of transmitting these credentials.

- **R5: The most commonly used credentials should be covered**.
- **R6: The most commonly used ways of sending the authentication information should be covered**.

Since 70% of the Web APIs send authentication information directly in the URI, while less than one third require that the HTTP header is constructed, initially providing support only for authentication via the URI might be sufficient.

The here described requirements will enable us to design a model capable of capturing commonly used authentication approaches as part of semantic Web API descriptions.

9.4 Web API Authentication Model

In this section we introduce the Web API Authentication (WAA) Model. In particular, we explain the design decisions that we have made and focus on describing the main parts of the model, which were introduced in order to support Web API authentication.

9.4.1 Design Decisions

In order to overcome the current heterogeneity of the authentication approaches and to provide a means for the automatic recognition and processing of authentication details, we introduce the Web API Authentication (WAA) Model.

The process of defining this ontology was guided by a number of competency questions. First, we started by identifying the cases, in which authentication is required, and the information that is needed. Relevant information in this respect is: *"Does the service require authentication?"*, *"Which operations require authentication?"*, *"What kind of authentication is used?"*, and *"What is the required information to complete the authentication?"*. As we concluded, based on the analysis of common authentication approaches, authentication has three main characteristic, including the required credentials, the used authentication protocol, and the way of sending the authentication information. Therefore, we can identify the information necessary for supporting a particular authentication mechanism by determining: *"What are the required credentials?"*, *"What is the used authentication protocol?"* and *"How is the authentication information transmitted?"*.

In addition to the competency questions, used for identifying the information that needs to be captured by the authentication ontology, we also followed some complementary design decisions. In particular, we advocate clarity, coherence, extensibility, minimal encoding bias and

minimal ontological commitment. We also take into consideration design decisions that support the reusability of the developed solution. For instance, the model should be modifiable to cover further authentication-relevant details. In addition, the model should be easy to interpret and use while generating annotations and, finally, the conceptualisation of the authentication information should be designed independently of the implementation solution in the form of an authentication engine.

In this way we ensure that the resulting ontology can capture the input authentication information, the way that is processed, and the way in which it is transmitted. In the following section we describe the Web API Authentication Model in detail.

9.4.2 Extending MSM with Authentication Support

In this section we describe the Web API Authentication (WAA) Model, providing details on each of the defined classes and properties. Furthermore, we provide annotation examples that demonstrate the use of WAA as part of semantic Web API descriptions. Figure 9.2 depicts the Web API Authentication Model with namespace *waa* (Web API Authentication), which consists of three main classes – *AuthenticationMechanism*, *Credentials* and *Service Authentication*[5].

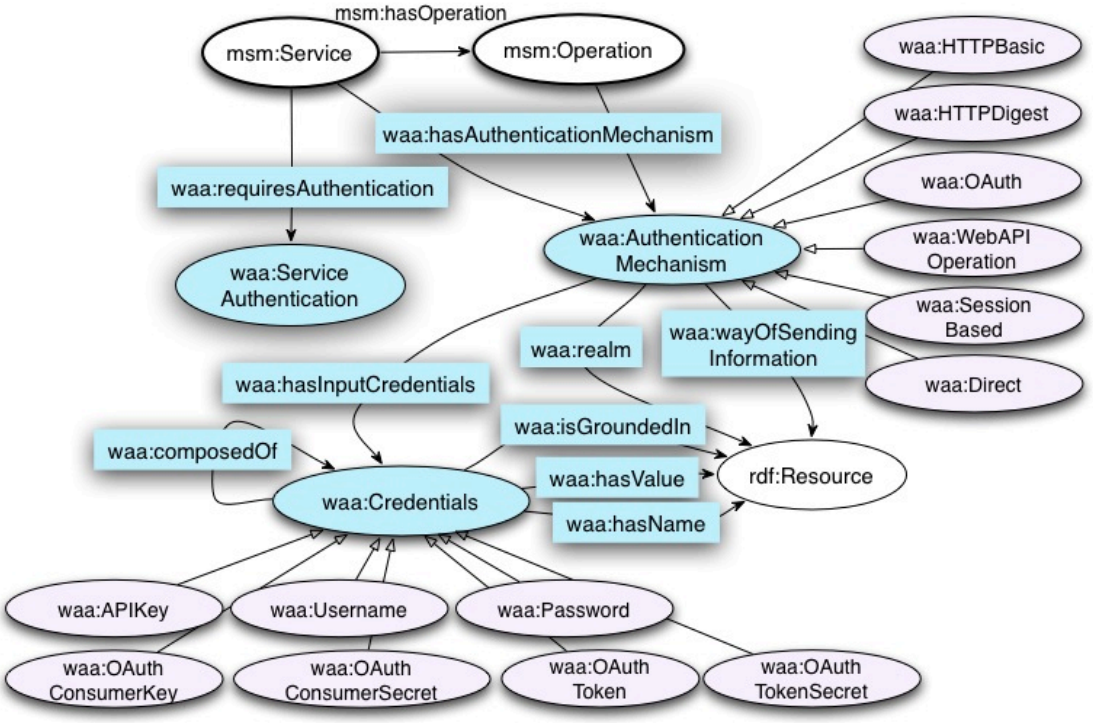


FIGURE 9.2: Web API Authentication Model

[5]The ontology is available at `http://purl.org/waa/`

The *AuthenticationMechanism* class is used to describe the way, in which the authentication is performed, including an implicit definition of the underlying protocols. It has six subclasses[6], corresponding to common authentication protocols, where the *Direct* subclass is used to describe approaches, which rely on using only credential details and employ no authentication protocol. The *HTTPBasic* and *HTTPDigest* classes prescribe not only the authentication mechanism but also indicate the subsequent communication for exchanging and confirming credentials, as prescribed by each of the protocols. The same applies to the *OAuth* class. In order to be able to indicate, which is the specific operation used for performing the authentication, we have introduced the *hasAuthenticationOperation* property of the *WebAPIOperation* class. It points directly to the operation URI. Finally, the *SessionBased* class is used to describe authentication based on the session information. This usually does not result in additional client-server communication, but its specification is important since this means that the communication with the API does not happen in a stateless manner. The list of authentication mechanism classes can be modified in order to include newly established approaches, such as Web-key, for example.

The *Credentials* class is used to capture the authentication credentials for performing the authentication. It has a number of subclasses including *APIKey*, *Username*, *Password*, *OAuthConsumerKey*, *OAuthConsumerSecret*, *OAuthToken* and *OAuthTokenSecret*, which can be combined (*waa:composedOf*) to produce composite credentials, such as authentication via username and password. These are predefined and included in the model because they are commonly used. Naturally, the list of credentials can be extended to cover further details, such as telephone number and pin, or email and password, or security token. Each credential, has a name and a value, which are used to capture the exact name of the credential, as used by the API, and the corresponding input value for performing the authentication. In addition, credentials have an *isGroundedIn* property that specifies the particular part of the HTTP request where the credentials are transmitted. This property has the same function as the *rest:isGroundedIn* but instead of using it on input data, we use it on authentication credentials. It is important to point out that this is not required for describing the type of authentication that the API uses but is necessary for completing the actual invocation by an invocation engine, e.g., OmniVoke. This is especially relevant, since we want to avoid defining lifting and lowering mapping for authentication credentials (for more details see Section 9.5.1).

The *AuthenticationMechanism* has a property *realm* that is used to define the scope or the part of the API, for which the authentication mechanism is applicable. This is done based on a URI definition. It is important to point out that *rdfs:isDefinedBy* (on *msm:Service*, not visualised in the figure), *rest:hasAddress* (on *msm:Service* and *msm:Operation*, not visualised in the figure) and *waa:realm* can all point to different URIs, since the API description URI is not necessarily the same as the invocation URI (the address). Similarly an API can have a series of invocation

[6]For the subclasses of both *AuthenticationMechanism* and *Credentials* we used different colour-coding.

URIs, while using the same authentication for all of them, which is specified through the definition of a common realm of the authentication mechanism (see Listing 9.1 for an example that demonstrates the necessity to explicitly define the different API URIs).

In addition, the *AuthenticationMechanism* has a property *wayOfSendingInformation*, which describes the part of the HTTP request, where the authentication details are sent. They can be sent as part of the URI, in the HTTP body or as an HTTP header parameter. Therefore, the *wayOfSendingInformation* points to *rdfs:Resource*, which can be an instance of the HTTP vocabulary [Con11] (http:Body or an instance of http:HeaderName) or a literal String (name of parameter in URI template). The two most common ways of sending credentials are via the HTTP header and via the URI, the HTTP body option is included for completeness. Furthermore, the authentication mechanism can be defined for each operation individually or for the service as a whole (*hasAuthenticationMechanism*). Some APIs require authentication only for the operations that perform data manipulation, while others use a uniform authentication approach throughout the service. In the case where both the service and the operation have authentication mechanisms defined, the operation definition overwrites the service one.

The *Service* class has a relationship to the *ServiceAuthentication* class, which has three instances including *All*, *Some* and *None* that are used to point out that the service requires authentication for all its operations for only some of them or for none of them. Theses instances can be assigned by the user but can also the deduced by using reasoning, based on the individual operation annotations.

The *Service* and *Operation* classes are currently marked with the *msm* namespace, however, they serve as placeholders that can be replaced by the service and operation elements of any service model may it be semantic, such as MSM and MicroWSMO/SA-REST, or not. In this way, the ontology provides flexibility and can be used as an extension to existing formalisms or independently from them. In the context of our work on supporting the automated invocation of Web APIs, we use it in conjunction with MSM and the previously defined Web API Grounding Model, in order to support the invocation of services, which require authentication.

Currently the Web API Authentication Model is in its second revision[7]. The first version of *waa* was published in [MPD+10b] and is available under `http://purl.oclc.org/NET/WebApiAuthentication`. The main changes include, the removal of the *TransmissionMedium* class, the addition of the *realm* property to the authentication mechanism, as well and the definition of *isGroundedIn*, *hasName* and *hasValue* properties on the *Credentials*.

In order to show how the authentication ontology can be used to annotate Web APIs, we have taken the Last.fm motivating example from Section 9.2 and provide its semantic descriptions (Listing 9.1). In the example we use the *msm* (`http://purl.org/msm`) namespace for

[7] `http://purl.org/waa`

the Minimal Service Model, as the underlying service model. Listing 9.1 shows the semantic description of the Last.fm API, including the authentication information. The Web API requires authentication for all its operations (Line 11) and has authentication information for the artist.getInfo operation reflected in Line 16. In particular the instance of the *Authentication-Mechanism* class contains details about the operation requiring an API key, which is sent in the URI without the use of any authentication protocols. As can be seen, capturing authentication information with the provided Web API authentication ontology is very simple and easy to apply.

```
1  @prefix : <http :// iserve.kmi.open.ac.uk/resource/services/e8f9548e-bbed-43fe-9d8a-71b7fdef9da#> .
2  @prefix rdf: <http :// www.w3.org/1999/02/22-rdf-syntax-ns#> .
3  @prefix rdfs: <http :// www.w3.org/2000/01/rdf-schema#> .
4  @prefix msm: <http://purl.org/msm#> .
5  @prefix rest: <http :// purl.org/hRESTS#> .
6  @prefix waa: <http://purl.org/waa#> .
7
8  :lastfmService a msm:Service ;
9     rdfs:isDefinedBy <http://www.last.fm/api/show?service=267> ;
10    rest:hasAddress "method=artist.getinfo&artist={p1} &api_key={p2}"^^rest:URITemplate ;
11    waa:requiresAuthentication waa:All ;
12    msm:hasOperation :ArtistGetInfo .
13 : ArtistGetInfo a msm:Operation ;
14       msm:hasInput :ArtistGetInfoInput ;
15       rest:hasAddress "http://ws.audioscrobbler.com/2.0/?"^^rest:URITemplate ;
16       waa:hasAuthenticationMechanism :lastfmAuth .
17 :lastfmAuth a waa:Direct ;
18       waa:realm <http://www.last.fm/api/> ;
19       waa:hasInputCredentials :api_key ;
20       waa:wayOfSendingInformation waa:ViaURI .
21 :api_key a waa:APIKey ;
22       waa:isGroundedIn "p2" ^^rdf:PlainLiteral .
23 : ArtistGetInfoInput a msm:MessageContent ;
24       msm:hasPart :artist .
25 : artist a msm:MessagePart ;
26       rest:isGroundedIn "p1"^^rdf:PlainLiteral .
```

LISTING 9.1: Example Service Description with Authentication Details

Similarly to MSM and the Web API Grounding Model, we also provide support for including the corresponding WAA model markup within the HTML documentation. We do this by including model references and not explicitly defining new hRESTS/MicroWSMO tags (see Appendix, Section A.3). In this way, the semantic descriptions do not have to be created directly, but instead can be generated in a bottom-up manner, by starting with the existing HTML documentation and syntactically structuring and enhancing it with the help of markup. Listing 9.2 shows the annotated HTML of the Last.fm API. The Web API requires authentication for all its operations (Line 2) and has authentication information for the *artist.getInfo* operation reflected in Line 4. The model reference contains a URI pointing to a particular instance of the *AuthenticationMechanism* class, which contains details about the operation requiring an API key, which is sent in the URI without the use of any authentication protocols.

```
1  <div class="service" id="service1"><h1>Last.fm Web Services</h1>
2  <a rel="model" href="http://purl.oclc.org/NET/WebApiAuthentication#All">
3  <div class="operation" id="op1"><h2><span class="label">artist.getInfo</span></h2>
4  <a rel="model" href="http://purl.oclc.org/NET/WebApiAuthentication/LastFm">
5  <span class="address">http://ws.audioscrobbler.com/2.0/?method=artist.getinfo...</span>
6  <div class="input" id="input1 ">...</ div>
7  <div class="output" id="output1">Artist</div></div></div>
```

LISTING 9.2: Annotating Authentication Information in the HTML Documentation

Listing 9.3 shows how this instance of the *AuthenticationMechanism* class looks like.

```
1  @prefix rdf:      <http :// www.w3.org/1999/02/22−rdf−syntax−ns#> .
2  @prefix waa:   <http :// purl.oclc.org/NET/WebApiAuthentication#> .
3  <http :// purl.oclc.org/NET/WebApiAuthentication/LastFm> rdf:type waa:Direct ;
4    waa:realm <http://www.last.fm/api/> ;
5    waa:hasInputCredentials <http://purl.oclc.org/NET/WebApiAuthentication/LastFmAPIKey> ;
6    waa:wayOfSendingInformation waa:ViaURI .
```

LISTING 9.3: Example Instance of the AuthenticationMechanism Class

Finally, listing 9.4 shows how example authentication *waa:APIKey* credentials can look like.

```
1  @prefix rdf:      <http :// www.w3.org/1999/02/22−rdf−syntax−ns#> .
2  @prefix waa:   <http :// purl.oclc.org/NET/WebApiAuthentication#> .
3  <http :// purl.oclc.org/NET/WebApiAuthentication/LastFmAPIKey> rdf:type waa:APIKey ;
4    waa:hasName "Last.fm API Key" ;
5    waa:hasValue "myAPIKey29813719823918273" .
```

LISTING 9.4: Example Credentials for APIKey

We differentiate between the API description instances published in a semantic Web service repository, such as iServe ([PLM+10]), and the ones used at invocation runtime. In particular, the HTML annotations would not include the values for the authentication instances. However, at runtime the description needs to actually include the authentication credentials as values in order to enable their processing by OmniVoke, as part of the invocation process. For example, the APIKey instance in line 20 (Listing 9.1) would have and actual key value attached via the *hasValue* properly (*:api_key a waa:APIKey; waa:hasValue "myAPIKey29813719823918273".*). Furthermore, we handle the lowering of the authentication credentials implicitly through the use of the *isGroundedIn* property, thus directly binding the input value to the part of the HTTP request where it is transmitted. Therefore, there is no need for defining extra lowering mappings for individual credentials. The remaining processing logic that is required for completing authentication as part of the invocation process is directly implemented in the invocation engine.

The Web API Authentication Model enables the orthogonal handling of the authentication credentials and the input. This is achieved by using the authentication model as an extension to service-description ontologies, by simply attaching it to the service and operation elements. This is important since one API can have a number of authentication options or an API might change the used authentication mechanism, while the underlying service description remains

the same. In order to be able to modify or exchange authentication-relevant parts of the description, without having to touch the remainder of the service description, we completely detach the definition of authentication-relevant details under the *waa* namespace.

9.5 Implementation

The Web API Authentication Model has been used in two invocation engines. These implementations are described in detail in the following section and exemplifying the usage of WAA as part of the API invocation process.

9.5.1 Authentication Engine Implementation

The Web API Authentication Model was initially used in SPICES[8] [AMG+10] (Semantic Platform for the Interaction and Consumption of Enriched Services). SPICES is developed by Guillermo Álvaro, Ivan Martínez from Intelligent Software Components (iSOCO) in Madrid, Spain, within the scope of the SOA4All project[9]. This implementation shows how the model can be successfully used to support the automated invocation of services. SPICES is a platform that enables the easy consumption of both WSDL-based services as well as Web APIs, by using their semantic descriptions. In particular, SPICES supports both the end-user interaction with services and the invocation process itself, via the generation of appropriate user interfaces. Typically, the user is presented with a set of fields, which must be completed to allow the service to execute, and these fields cover input parameters as well as authentication credentials.

Dealing with the different types of credentials and the way they have to be used, is the purpose of an *Authentication Engine*, which is part of SPICES and is developed as a REST service, capable of handling the storage and retrieval of credentials for different Web APIs. This engine has the necessary logic to support the user in his/her interaction with services. In particular, if the engine has the credentials for a given service, thanks to the authentication annotations described previously, it is able to create a suitable request including the credentials. If the authentication credentials are not available yet, based on the authentication annotations, the authentication engine is aware of the missing credentials and will prompt the user to provide them. Figure 9.3 shows how the authentication engine prompts the user for the Last.fm API key, based on the API annotation, during the process of invoking the *artist.getInfo* operation. In this way the authentication engine can collect the required credentials and compose an API request, which together with the provided input information, supports the automated Web API invocation.

[8] `http://soa4all.isoco.net/spices`
[9] `http://cordis.europa.eu/project/rcn/85536_en.html`

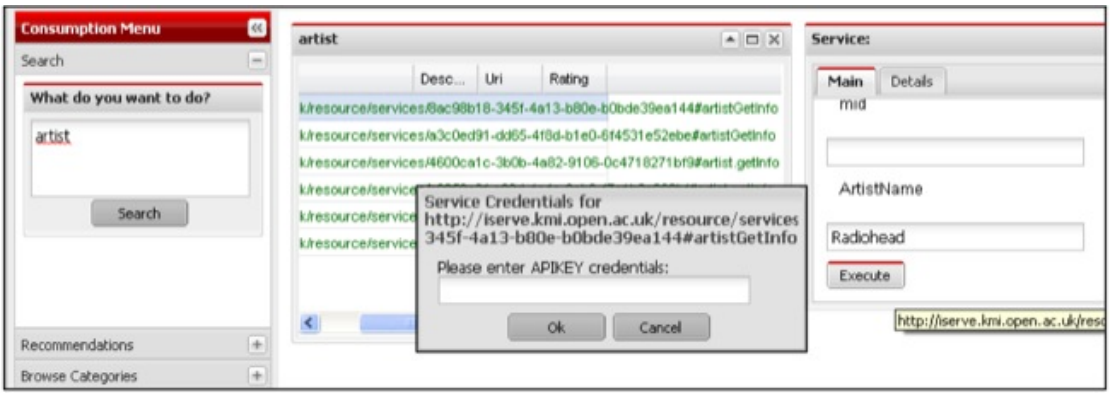


FIGURE 9.3: Invoking the Last.fm API

Currently, the authentication engine plays the role of a trusted party, since it accesses and stores all user credentials. However, SPICES is only a prototype application, with the main focus on supporting the invocation of Web APIs, and the authentication engine represents only an initial implementation of our approach. Therefore, some issues such as appropriately storing and managing user credentials, still need to be addressed.

SPICES used the first version of the Web API Authentication Model [MPD+10b]. Therefore, it serves as a proof-of-concept of enabling the integration of authentication as part of the automated invocation process. However, it had some limitations, since it relied on the direct mapping between the names of the input parameters and the invocation URL, and input data and authentication credentials were handled in the same way. More advanced support, based on the currently revised WAA model, is provided by OmniVoke [LMKD11].

As already mentioned, even if authentication is an almost indispensable part of the Web APIs invocation, when modelling, it has been intentionally kept separate from other API service properties such as the input description. However, when it comes to the practical implementation of the devised approach, the separation is not always apparent. As exemplified by the Last.fm API with invocation address `http://ws.audioscrobbler.com/2.0/?method=artist.getevents&artist=Cher&api_key=b25b959554ed76058ac220b7b2e0a026`, the API Key credential is used in the same way as other input parameters, such as the artist.

OmniVoke explicitly treats input data and credentials separately, since the way these are supplied, in order to invoke the actual Web API, is usually different. OmniVoke exploits semantic descriptions of service properties, including the data being processed by the service. Therefore, the input data passed to OmniVoke is in a semantic representation format, such as RDF, which could, for instance, be obtained dynamically from the Linked Data cloud via SPARQL queries. In contrast, authentication credentials are confidential and user-specific, usually provided manually by the user at invocation time, or stored in dedicated secure infrastructure. In addition, some authentication mechanisms require the completion of negotiation and verification protocol

steps. In order to facilitate this, OmniVoke uses the authentication part of the semantic description in order to provide an adequate user interface for acquiring the necessary credentials. In some cases, such as OAuth, this includes bringing up a web browser for accessing the service provider's authorisation page.

In summary, OmniVoke has the necessary logic to support the user in his/her interaction with services, including the processing of authentication. Given the Web API semantic description, OmniVoke is able to create an authenticated request to the actual Web API. In particular, if the authentication credentials are not available, based on the authentication annotations, OmniVoke will prompt the user to provide them.

Currently, OmniVoke's main focus is on supporting the invocation of Web APIs. Similarly to SPICES, the handling of authentication credentials by OmniVoke raises issues such as appropriately storing and managing user credentials. However, since more than 70% of the authentication approaches do not protect the user credentials (for example, through encryption of the password or key), this issue is not that crucial. Nevertheless, possible solutions include that the user credentials are only cached for the duration of the invocation, in order to guarantee their privacy and validity. As an alternative, OmniVoke could implement its own OAuth service so that the sent credentials are protected. Both SPICES and OmniVoke demonstrate that the task of providing a unified support for handling authentication as part of invocation is a challenging one. Still they also prove the practical applicability of the Web API Authentication Model.

9.6 Summary

In the context of Web APIs and in contrast to traditional Web services, security is essentially limited to authentication. Currently none of the existing approaches for describing Web APIs support the automated authentication as part of the Web API invocation process. As a result, developers are required to manually retrieve and interpret the HTML Web API documentation, to signup with API providers, in order to receive access credentials, and to implement support for the different authentication protocols. In addition, none of the existing frameworks for supporting the creation of mashups, such as Yahoo Pipes and DERI Pipes, enable the handling of authentication in an integrated way, and it has to be addressed with additional manual effort.

Our Web API studies show that more than 80% of the APIs require authentication for invoking an operation, which makes it a vital part of the invocation process and any invocation approach disregarding authentication information has very limited support. Therefore, we propose the annotation of authentication information by using the Web API Authentication Model, which overcomes Web API description heterogeneity and provides the basis for its automated handling as part of the invocation process. We base the annotation approach on a thorough study of current

Web API authentication mechanisms and have determined the most commonly used ones. We determine their characteristics, in order to be able to describe them, and compose a set of design principles and requirements. Based on this analysis, we develop a Web API Authentication Model (WAA) that is capable of capturing the necessary details for supporting authentication as part of the invocation process.

Chapter 10

Supporting the Creation of Semantic Web API Descriptions

The previous chapters introduced means for modelling, annotating and invoking Web APIs by providing a description model and specific extensions that capture common service characteristics and lay the foundation for automating API use, especially by focusing on invocation and authentication support. However, in order to ease the adoption of the devised solutions it is necessary to provide at least some level of support for developers. Therefore, this chapter describes in detail the support for adopting the introduced contributions towards a more automated Web API use in terms of tools, as well as guidelines on how to create Web API descriptions, incorporating automation of some of the commonly performed annotation tasks such as determining the type of service.

10.1 Introduction

MSM, together with the Web API Grounding Model and the Web API Authentication Model, can be used as a basis for making HTML service documentation machine-interpretable and enable the adding of semantic information, thus providing the means for creating semantic descriptions of Web APIs. However, without supporting tools and guidelines, developers would have to modify and enhance the documentation manually by using a simple text/HTML editor. In addition, the complete annotation process would have to be done manually, if there are no tools, which enable the search for suitable domain ontologies or the reuse of annotations of previously semantically described services. In order to address these challenges and to facilitate the easier adoption of the models for semantically describing Web APIs, we introduce SWEET: Semantic Web sErvices Editing Tool[1], and provide two approaches for automating one of the

[1] `http://sweet.kmi.open.ac.uk`

most frequently performed and used annotation types – determining the kind of functionality that the service provides.

SWEET is developed as a Web application, that can be launched in a common Web browser and does not require any installation or additional configuration. It provides key functionalities for modifying the HTML Web API documentation in order to include markup that identifies the different parts of the API, such as operations, inputs and outputs, and also supports adding semantic annotations by linking the different service parts to semantic entities. As a result, SWEET enables the creation of complete semantic Web API descriptions, based on the previously introduced models, given only the existing HTML documentation. More importantly, the tool hides formalism and annotation complexities from the user by simply visualising and highlighting the parts of the API that are already annotated and produces HTML documentation that is visually equivalent to the original one but is enhanced with metadata that captures the syntactical and semantic details of the API. The resulting HTML documentation also serves as the basis for extracting an RDF-based semantic Web API description, which can be published and shared in a service repository, such as iServe [PLM+10], enabling service browsing and search. In the following sections we introduce the different consecutive versions of SWEET, which vary in the level of annotation support that they provide and use models with different level of expressivity.

Still, even with the help of SWEET, the creation of semantic Web API descriptions remains a somewhat effort consuming task, especially in the context of finding suitable ontologies and annotations for enhancing the API. Therefore, in order to ease the process of providing semantic service metadata, this chapter describes a number of simple approaches that target the automation of some of the key tasks such as ontology search and API functionality classification. In particular, we provide an integrated way for directly retrieving a list of suitable domain ontologies and include two classification approaches, one of which addresses the multi-lingual context of the API documentation. Since Web APIs are commonly described in human-oriented documentation such as webpages, they are not necessarily in English, therefore, it is important to be able to determine the type of service, independently of the original language of the description. The tool, as well as the supporting annotation task automation approaches, is thoroughly evaluated.

This Chapter is structured as follows: Section 10.2 introduces SWEET, describing its main functionalities, the provided annotation support as well as the development of the tool in its different versions. Section 10.3 focuses on the solutions that provide automation to individual annotation tasks, thus contributing to the semi-automated creation of semantic Web API descriptions. These include ontology and annotation search support as well as classification of the APIs, based on the functionality that they provide. Finally, Section 10.4 concludes this chapter. The evaluation of the tool and the corresponding approaches in presented in Chapter 11.

10.2 SWEET

SWEET is the first tool developed for supporting users in creating semantic Web API descriptions by structuring the HTML documentation and associating semantic annotations. SWEET[2] is a Web application developed using JavaScript and ExtGWT[3]. It is part of a fully-fledged framework, developed within the scope of the SOA4All European project (SOA4All EU project FP7 - 215219, `http://cordis.europa.eu/project/rcn/85536_en.html`), for supporting the lifecycle of services, particularly targeted at enabling the creation of semantic Web API descriptions. SWEET takes as input an HTML Web page documenting a Web API and offers functionalities for annotating service properties and linking semantic information to them. The annotations can be created by the API providers or anyone interested in enhancing the documentation with metadata.

SWEET uses the hRESTS-based microformat HTML tags for the creation of machine-readable service descriptions and supports the semantic annotation of service properties, as specified in the semantic models introduced in this thesis. A current version of the tool can be found under `http://sweetdemo.kmi.open.ac.uk/soa4all/MicroWSMOeditor.html`, while all code, including previous and latest releases, are in GitHub[4] under `https://github.com/kmi/SWEET-ExtGWT`.

10.2.1 Design and Architecture

As introduced earlier, the vast majority of available Web APIs are described in HTML, directly as part of webpages. Therefore, we advocate using microformats for structuring service documentations and for attaching semantic annotations to them. This is a lightweight non-invasive approach that does not require the creation of complex API descriptions from scratch but rather enhances the already existing documentation and relies on employing a few semantic annotations for supporting the automation of commonly performed service tasks, instead of providing extensive reasoning and deduction support. This approach is reflected in the previously described models and is also implemented in SWEET.

With the help of SWEET, API providers and developers alike can just navigate to the Web API documentation and annotate it. Providers can replace the documentation with the enriched version and share it, easing subsequent processing and implementation. Developers can keep the annotated documentation locally or share it on a dedicate repository, where it can be reused by other developers.

[2] `http://sweet.kmi.open.ac.uk`
[3] `http://extjs.com/products/gxt/`
[4] `https://github.com`

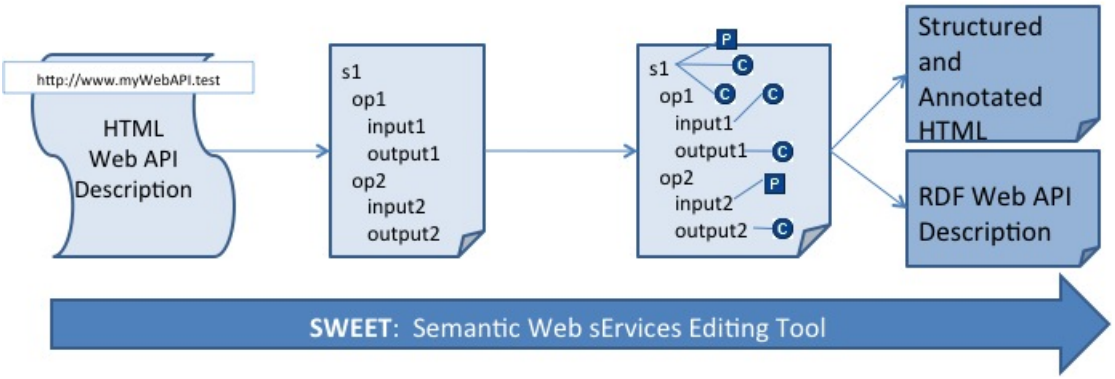


FIGURE 10.1: Semantic Annotation of Web APIs

The creation of semantic Web API descriptions is based on a 3-step process visualised in Figure 10.1, which is also the way, in which SWEET supports API annotation. First, the unstructured text documentation of the Web API, in the form of HTML content, is extended with hRESTS tags, such as the ones defined via mappings to MSM, the Web API Grounding Model and the Web API Authentication Model, marking all service properties. The resulting service structure is enriched with semantic annotations by adding links pointing to semantic entities, by adopting the SAWSDL approach for adding model references. Finally, the annotated HTML can be saved and republished, or it can be used to extract RDF-based semantic descriptions. The user is supported in completing each of these steps by SWEET, which guides him/her in the process of creating annotations and hides formalism complexities behind an easy-to use interface.

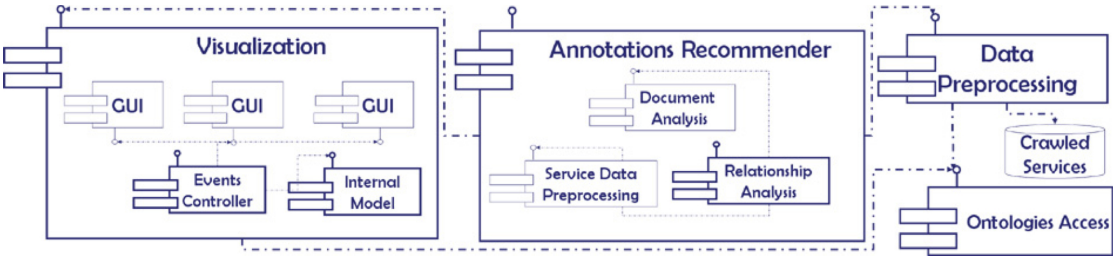


FIGURE 10.2: SWEET Architecture

SWEET is designed as a classical three-layered Web application. As can be seen in Figure 10.2, the architecture of SWEET consists of three main components, including the visualisation component, the data preprocessing component and the annotations recommender. The visualisation component is based on a model-view-controller (MVC) [GHJV94] architecture design pattern, where the model implements an internal representation of the annotated Web API, in accordance with the elements foreseen by the semantic formalisms detailed in the previous chapters. In this way, every time the user adds a new annotation via the interface, the model representation of the Web API description is automatically updated. Similarly, when parts of the model representation are altered or deleted, the highlighting and visualisation in the user interface is also

adjusted. When the annotation process is completed, the resulting HTML and RDF Web API descriptions are generated based on the produced internal model.

The GUI of the visualisation component is shown in Figure 10.3 and it has three main panels. The HTML of the Web API is loaded in the *Navigator* panel. Based on this, the HTML DOM of the APIs can freely be manipulated by using functionalities of the *Annotation Editor* panel. The current status of the annotation is visualised in the form of a tree structure in the *Semantic Description* panel, which is implemented by automatically synchronising the visualisation of the service annotation with an internal model representation, every time the user manipulates it.

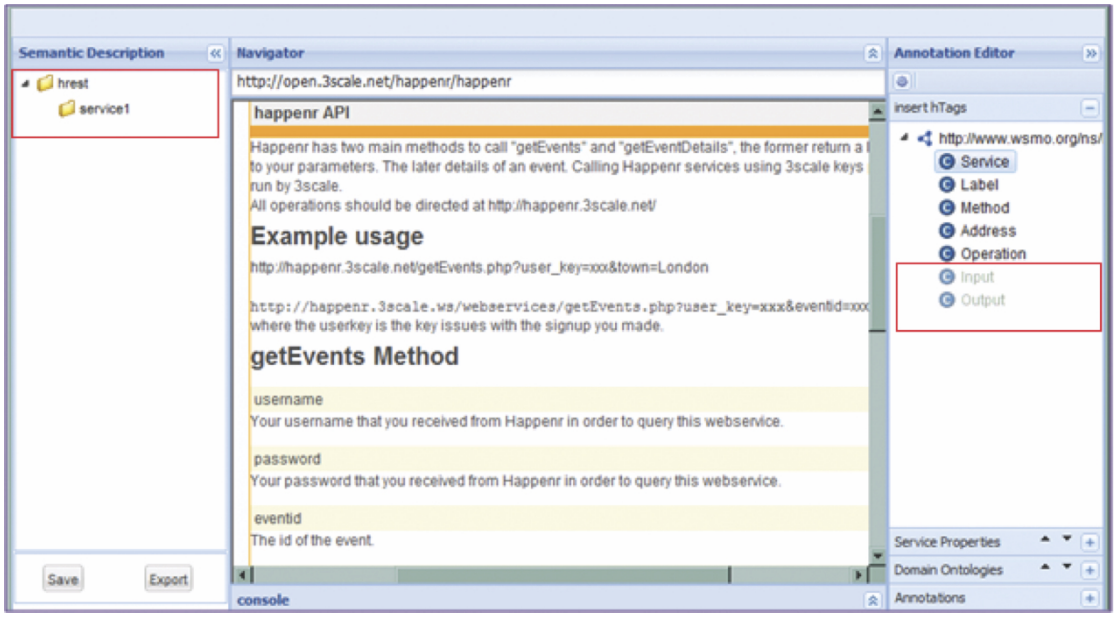


FIGURE 10.3: SWEET: Inserting hRESTS Tags

The annotations recommender component assists the user in annotating a service, by suggesting suitable annotations for the service as a whole (domain ontology recommendation) and for its individual properties. This component includes two main functionalities, namely, support for ontology search and classification of the type of Web API. These are discussed in more detail in Section 10.3. Finally, the data preprocessing component implements functionalities for data preparation for the visualisation component, caching mechanisms and simple rule-based analysis.

All components include points of extensibility. Overall, SWEET effectively supports users in creating semantic Web API descriptions by marking service properties, searching for suitable ontologies, and linking semantic information. MSM is already fully supported by the current version of SWEET. The existing annotation process can be further automated by extending the recommender with additional support.

When the user completes the semantic annotation of the HTML documentation, the annotated HTML can be saved and republished on the Web, representing an instance of a semantic Web

API. Similarly, the resulting HTML can also be transformed into an RDF-based description, which can be used as the basis for performing common tasks such as search, composition and invocation. Currently, all created descriptions can be directly posted to iServe, where they can be browsed and searched alongside previously annotated Web APIs but also together with semantic WSDL-based descriptions.

10.2.2 SWEET Bookmarklet

SWEET's development underwent a number of versions, each one featuring extended functionalities and updates in accordance with the changes to the semantic service models. Therefore, the development of the tool and of the semantic formalisms happened in an iterative way, where new elements in the models resulted in adding the corresponding support in SWEET, while user-feedback of the tool resulted in some cases in necessary adjustments to the models.

The first version of SWEET is in the form of a bookmarklet, which can be loaded on top of any webpage, and, in particular, on top of the HTML Web API documentation. It is a JavaScript Web application, which requires no installation and has the form of a vertical widget, which appears on top of the currently browsed webpage. This tool overcomes a number of difficulties associated with the annotation of Web APIs, including the fact that the HTML documentation is viewed through a browser and that the user who wants to annotate the service webpage, usually does not have access to manipulate or change the HTML.

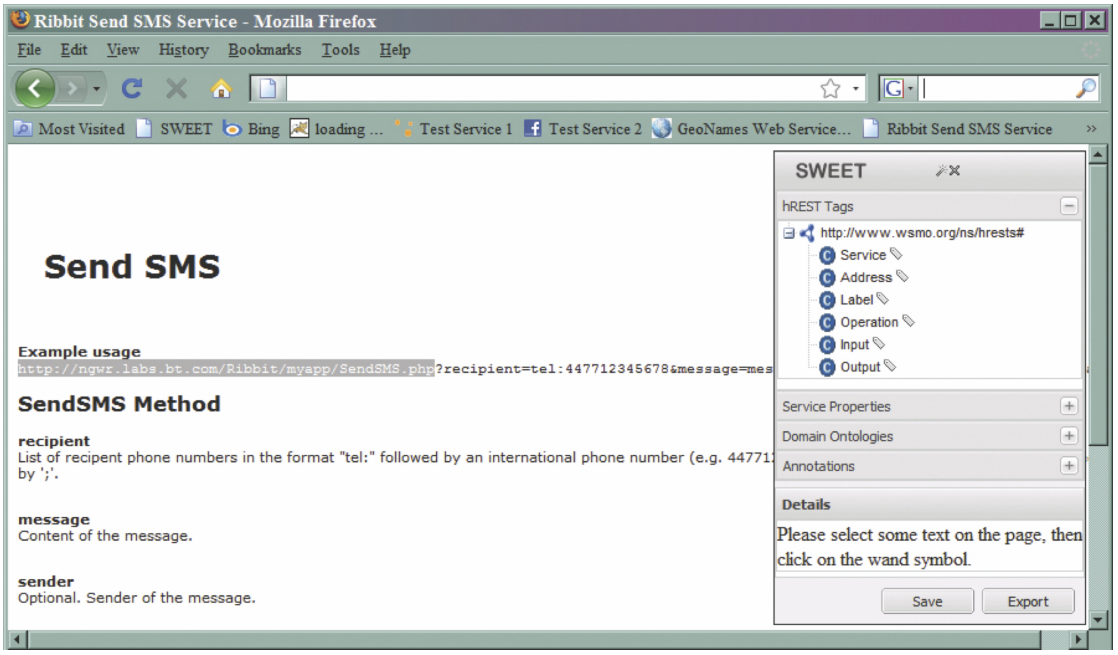


FIGURE 10.4: SWEET: hRESTS Annotation

Figure 10.4 shows a screenshot of the SWEET bookmarklet. The first version of SWEET sticks strictly to the defined hRESTS tags, while the following versions include extensions based on

MSM, the Grounding Model and WAA. In particular the use of microformats facilitate the translation of the HTML tag structure into objects and properties. Therefore, the visualisation of the HTML documentation remains unchanged, while the microformat uses `class` and `rel` XHTML attributes to mark key service properties. The hRESTS tags can simply be inserted by selecting the relevant part of the HTML and clicking on the corresponding class node in the hRESTS panel of SWEET. In this way, the hRESTS annotation process is less error-prone, less time-consuming and much simpler for the user. The result is an annotated copy of the original HTML, which can easily be converted into RDF ("Export" button), by using an implemented XSLT stylesheet.

```
1  <div class="service" id="svc">
2  <h1>Send <span class="label">SMS Service</span></h1>
3  <p>This is a Short Message Service (SMS).<p>
4  <b>Example usage</b>
5  <span class="address">http://my.test.serv.com/SendSMS.php?recipient=
6   tel:447712345678&message=messagetext&sender=User&title=TheMessage</span>
7  <div class="operation" id="op1">
8  <h2><code class="label">SendSMS Method</code></h2>
9  <span class="input">
10 <b>recipient</b><p>List of recipient phone numbers in the format "tel:"
11 followed by an international phone number</p><br/>
12 <b>message</b><p>Content of the message.</p></span><br/>
13 <h2><span class="output">The result is a sent SMS.
14 </span></h2></div></div>
15
```

LISTING 10.1: Example Web API Description with hRESTS Tags

Listing 10.1 shows a simple description example, annotated with hRESTS by using the SWEET bookmarklet. It visualises the usage of the microformat annotations, as well as the structure restrictions, which exist for the different classes. The complete API description is marked by the `service` class. The service may have a `label`, which can be used to mark the human-readable name of the service. A machine readable description can be created, even if there is no service class inserted. It is sufficient that the HTML description contains at least one operation, which is marked with the `operation` class. The operation description itself includes the `address` where it can be executed and the HTTP `method`.

The final two elements are `input` and `output`. They are used on block markup and indicate the operation's input and output. These two elements serve as the basis for extensions given by microformats, which provide additional semantic information or details about the particular datatype and schema information. Finally, user-oriented labels can be attached to the service, operation, input or output classes.

The microformat markup that can be generated with the help of the first version of SWEET sticks strictly to the original hRESTS specification. Therefore, the resulting syntactic structuring of the HTML documentation provides relatively limited support for using Web APIs, for example, because parts of the input and output cannot be specified, while the invocation address and the grounding of the parameters in the different HTTP message parts is missing completely.

Therefore, the produced descriptions enable, to a certain extent, the completion of tasks such as search based on the inputs and outputs. However, they are not suitable for composing and invoking Web APIs, since key information such as the individual input and output parts, which are necessary in order to realise the data transformation between two consecutive APIs, or identifying parameters transmitted as part of the URL, which is needed for completing an API call, are simply not captured.

The HTML, with the inserted hRESTS microformat tags, can be enriched with semantic metadata by adopting the SAWSDL approach. SAWSDL specifies three XML attributes with equivalent RDF properties, including `modelReference`, `liftingSchemaMapping` and `loweringSchemaMapping`. The first version of SWEET supports only adding model references, while the links to lifting and lowering schema definitions need to be added manually, directly in the generated HTML file.

The task of associating semantic content with the Web API properties is even more time- and effort-demanding than the insertion of hRESTS tags. Therefore, SWEET supports users in searching for suitable domain ontologies and in making semantic annotations. Whenever, a user wants to add semantics to a particular service property, for example, an input parameter, he/she has to select it and click on the "magic wand" symbol, which sends a request to Watson [dSM+08]. Watson is a search engine, which retrieves relevant ontologies based on keyword search.

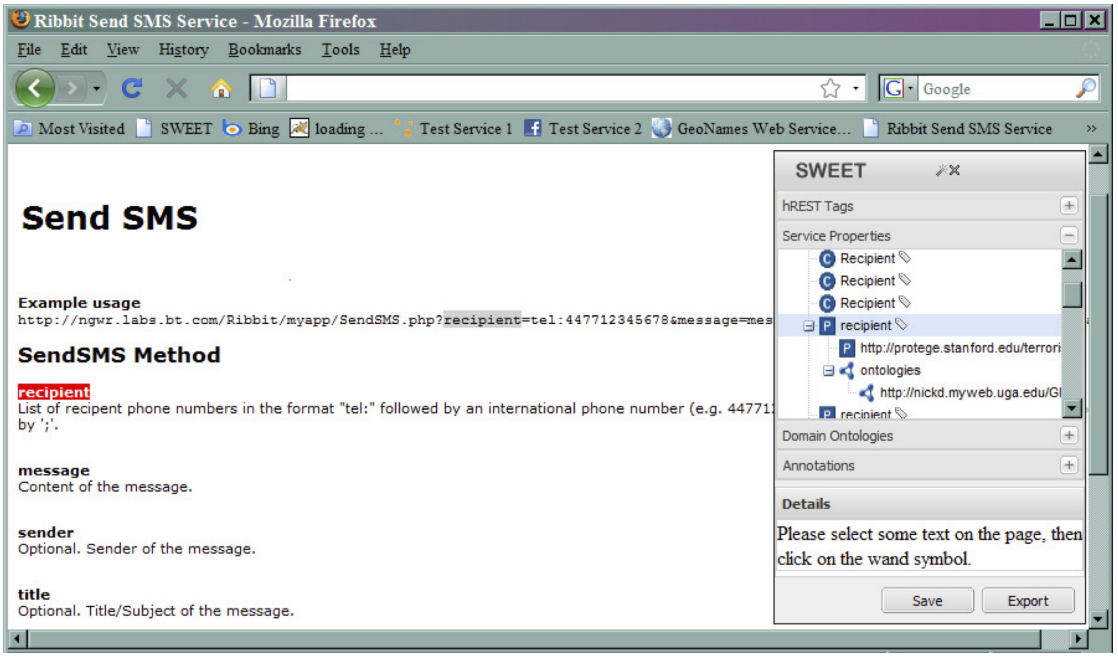


FIGURE 10.5: SWEET: Semantic Annotation

The results are presented in the *Service Properties* panel visualised in Figure 10.5. The searched for property is the root of the tree, populated with nodes that represent the found matches.

In the example, *recipient* was found to be a property ("P") in an ontology located at `http://protege.stanford.edu`. If the user needs additional information in order to decide whether the particular semantic annotation is suitable or not, he/she can switch to the *Domain Ontologies* panel, which provides a list of all concepts and the corresponding property matches. The user can make a semantic annotation by simply selecting the property instance and clicking on one of the semantic matches in the *Service Properties* panel. The result is an annotated HTML document, including hRESTS tags and model references, and it can be saved in a repository (button "Save") or be converted into RDF (button "Export").

Listing 10.2 shows the example Web API documentation annotated with the first version of SWEET. Line 4 uses the `model` relation to indicate that the service sends SMS, while line 12 associates the input parameter *recipient* with the class *Recipient*. The lowering schema for the recipient is also provided in line 13 but this link has to be added manually and is not supported by the initial version of the tool.

```
1  <div class="service" id="svc">
2  <h1>Send <span class="label">SMS Service</span></h1>
3  <p>This is a
4  <a rel="model" href="http://example.com/telecommunications/sendSMS">
5  Short Message Service (SMS).</a><p>
6  <b>Example usage</b>
7  <span class="address">http://my.test.serv.com/SendSMS.php?recipient=
8  tel:447712345678&message=messagetext&sender=User&title=TheMessage</span>
9  <div class="operation" id="op1">
10 <h2><code class="label">SendSMS Method</code></h2>
11 <span class="input">
12 <b><a rel="model" href="http://example.com/data/onto.owl#Recipient">
13 recipient</a>(<a rel="lowering" href="http://example.com/data/sms.xsparql">
14 lowering</a>)</b><p>List of recipient phone numbers in the format "tel:"
15 followed by an international phone number</p><br/></div></div>
```

LISTING 10.2: Example Web API Description with Model References

In summary, the SWEET bookmarklet provides support for three main actions – adding hRESTS tags, integrated ontology search, and adding model references. It offers the basic functionalities required for annotating Web APIs and represents the first contribution towards supporting the creation of semantic Web API descriptions. Based on the gathered experience as well as the user input, the bookmarklet was reimplemented as a stand-alone Web application described in more detail in the next section.

10.2.3 SWEET Web Application

The initial concept of loading a tool on top of the webpage describing the Web API had to be adjusted for two main reasons. First, if the tool takes the form of a widget, the size of the user interface has to be relatively small so that the underlying API documentation is not covered. This means that all the annotation functionalities have to be visualised in a relatively constrained area, resulting in a lot of tabs or panels that make the user interface of the bookmarklet cluttered

and confusing. Second, the technology used to implement the bookmarklet is based on cross-domain communication, so that requests through the original HTML documentation website are redirected through the tool. This presents an insecure way of handing client-server communication and had to be abandoned as a solution approach. As a result, a new version of SWEET was implemented, which takes the form of a web application, where the HTML documentation is simply loaded into one of the interface panels.

The web application version of SWEET benefits from the user experience and feedback gathered through the bookmarklet release and enables the completion of the annotation process in a step-by-step manner. In particular, it is based on a set of simple requirements that were determined based on the needs, resulting from tasks involved in the creation of semantic Web API descriptions, but also taking into consideration the gathered user input. These requirements are as follows:

- **R1: Each of the annotation tasks should be presented in an individual visualisation unit**, such as a tab or a panel, that can be hidden once the particular task is completed.
- **R2: Functionalities, needed for the different annotation tasks, should not be intertwined**, both implementation- and visualisation-wise.
- **R3: It should be clearly visible what annotations are already made** and how far along the annotation process the user has come.
- **R4: Sufficient browsing options need to be presented**, so that there is enough information in order to be able to decide if a semantic entity is suitable to be used as an annotation or not.
- **R5: Work in progress should be saveable and retrievable** for later completion.
- **R6: The goal of the tool is to guide the user** but not to validate all the annotation steps that are made.

Requirements 1 and 2 mainly relate to the way the implementation and the user interface are realised. They are important in order to provide clear presentation and separation of the individual functionalities. As can be seen in Figure **??** this was achieved with the help of stacked panels, where each one can be shown or hidden, from top to bottom, while creating hRESTS annotation, searching for suitable ontologies, etc. Requirement 3 resulted in the *Semantic Description* panel that shows the parts of the description that have already been created. This is combined with corresponding borders and shading in the loaded HTML documentation. Requirement 4 was indirectly satisfied by enabling the saving of enhanced HTML documentation at any point of the annotation process and supporting its loading back into the tool. Therefore, it is recommended that only complete descriptions are posted to iServe, while work in progress is stored locally.

Regarding the last point, we found out that it is easiest to create semantic Web API descriptions in a step-by-step manner. Therefore, SWEET takes as input the Web API HTML webpage, which is loaded in the central panel, and returns a semantically annotated version of the HTML or a RDF semantic description. In order to do this the user needs to complete the following four mains steps:

1. Identify service properties (service, operation, address, HTTP method, input, output and label).
2. Search for domain ontologies suitable for annotating the service properties.
3. Annotate service properties with semantic information.
4. Save or export the annotated Web API.

The first step can easily be completed by simply selecting the part of the HTML, which describes a particular service property, and clicking on the corresponding tag in the *insert hTags* pane. The new feature here is that at the beginning, only the *Service* node of the hRESTS tree is enabled. After the user marks the body of the service, additional tags, such as *Operation* and *Method*, are enabled. In this way, the user is guided through the process of structuring the service description and is prevented from making annotation mistakes. The marking of HTML content with a particular hRESTS tag results in the insertion of a corresponding class HTML attribute. The formalism complexity is hidden from the user, and instead, he/she only sees the current status of the annotations, reflected in the *Semantic Description* panel. In addition, each inserted tag is highlighted by a custom cascading style sheet (CSS), which visualises the annotations the user has made.

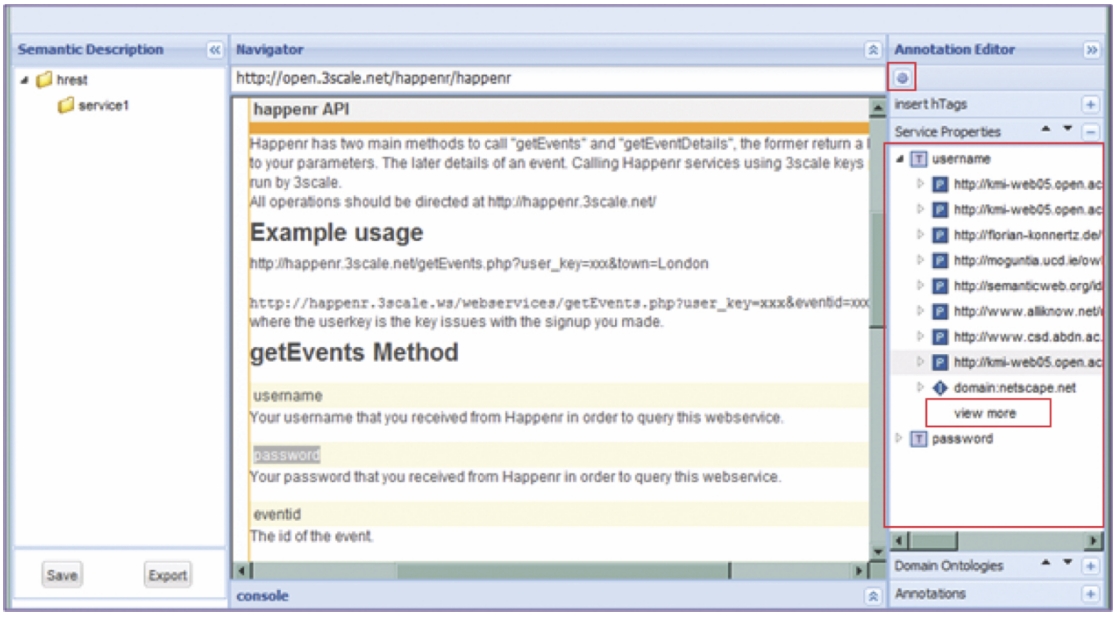


FIGURE 10.6: SWEET: Searching for Suitable Ontologies

After the user structures the HTML documentation and identifies all service properties, the adding of semantic information can begin. The new version of SWEET, just like the bookmarklet, supports users in searching for suitable domain ontologies by providing an integrated search with Watson [dSM+08]. The search is done by selecting a service property and sending it as a search request to Watson. The result is a set of ontology entities, matching the service property search, which are displayed in the *Service Properties* panel visualised in Figure 10.6. The main difference here is that there are more options for browsing and exploring the returned results, so that the user can decide, which annotation is most suitable to use. For shorter response times, only the first 20 matches are retrieved. If the first set of ontology results is insufficient, the user can search for more results by clicking on *view more*. In addition, the search is session-based and the user preserves his/her ontology search while annotating different service descriptions.

The implementation of the *Service Properties* and *Domain Ontologies* panels supports the user in choosing a suitable ontology for annotating the individual service properties or the complete Web API. These supporting functionalities are visualised in Figure 10.7. The user can view the URI of each of the matching concepts, properties or instances and the corresponding ontology. Additional information is available in the *Domain Ontologies* panel, which shows all service properties that can be annotated with one particular ontology as well as a list of all concepts. The entries of both panels can be expanded or collapsed in order to ease the navigation.

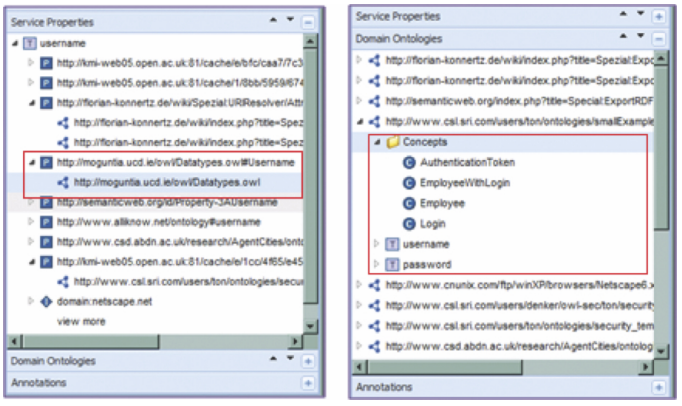


FIGURE 10.7: SWEET: Exploring Domain Ontologies

Once the user has decided, which ontology to use for the service property annotation, he/she can do an annotation by selecting a part of the service HTML description and clicking on *Semantic Annotation* in the *Service Properties* context menu. This results in inserting a `model` attribute and a reference pointing to the URI, of the linked semantic entity. MicroWSMO also contains elements for `lifting` and `lowering`, which point to links for lifting and lowering transformations. These can be added by inserting the URIs of the transformation files.

The result is a semantically annotated HTML description, with inserted `model` and `href` tags marking the association of the particular HTML elements with the semantic entities. A summary of the already made annotations is given in the *Annotations* panel. These annotations can be removed by choosing *Delete* from the context menu. In this way, the user can remove incorrect annotations and substitute them with new ones, without having to reload the tool and start the annotation process from the very beginning. An example of an annotated Web API description is given in the Appendix in Section A.1.

The web application version of SWEET also provides options for customising the way service descriptions are viewed. First, if the *Navigator* panel displays HTML service documentation, which already contain annotation elements, these elements will be recognised and automatically highlighted so that the user can manipulate them and integrate them in his/her own annotation of the Web API. Second, the way the service properties and semantic information is highlighted can be modified by simply substituting the current CSS file with a new one, which uses a different style. In addition, this version of SWEET implements the following new functionalities:

1. Renaming of service properties in the *Semantic Description* panel.
2. Deleting of service properties in the *Semantic Description* panel.
3. Adding of model references directly to different service parts.
4. Adding of lifting and lowering schema.
5. Definition of parameters as part of inputs and outputs.

Detailed descriptions of how to use SWEET, in its different versions, as well as short screen casts are available in Appendix B and at `http://sweet.kmi.open.ac.uk/download.html` and `http://sweet.kmi.open.ac.uk/demo.html` correspondingly. The tool was also used in a number to training sessions and tutorials[5]. Relevant publications include [MKP09, MPD09b] and [MGPD09]. The following section described the approaches used to evaluate SWEET.

10.3 Automating the Creation of Semantic Web API Descriptions

While SWEET enables the creation of semantic Web API descriptions based on the existing HTML documentation, the annotation process still requires quite some amount of manual work. In addition to marking service properties, users are also required to choose appropriate semantic

[5]The training materials available are at http://www.slideshare.net/mmaleshkova/automating-the-use-of-web-apis-through-lightweight-semantics, http://www.slideshare.net/mmaleshkova/handson-automating-the-use-of-web-apis-through-lightweight-semantics

annotations. While SWEET provides much more support than a simple text editor would, there is still potential for automating some of the annotation tasks, thus enabling a semi-automated process of creating semantic Web API descriptions.

In this section, we describe the suggested approaches for adding automation support to SWEET. In particular, we describe how the search for suitable ontologies used to semantically enhance Web APIs is realised and how the type of functionality that the API provides is automatically determined. The focus of the automation support is precisely on these two tasks because they play a substantial role during the process of creating semantic Web API descriptions. In particular, very few of SWEET's users would directly know, exactly which semantic entities they want to use to annotate the individual service properties. Therefore, instead of having to rely on external search tools or ontology directories, ontology search is provided as part of SWEET in an integrated way. Furthermore, including the type of functionality that the Web API provides as part of the semantic description is key for supporting common tasks such as discovery and composition, which are predominantly based on finding and using an API with the required support.

The following sections describe in more detail the contributions towards providing semi-automated support for creating semantic Web API descriptions, focusing in particular on annotation search and functionality classification solutions.

10.3.1 Annotation Search

Finding and determining the correct semantic entity that is to be used for making an annotation can be a difficult and time-consuming task, especially for someone not particularly familiar with a specific domain or with using semantic metadata in general. In order to assist users of SWEET in completing this task, the tool provides an integrated ontology search through Watson [dSM$^+$08, dM11]. Watson has three main functions: collecting the available semantic content on the Web, analysing it to extract useful metadata and indexes, and, finally, enabling search over the gathered semantic content. In the context of supporting the semi-automated annotation of Web APIs, the search functionalities provided by Watson are a key feature, especially since they are accessible directly over an API. Given a keyword, or a set of keywords, Watson returns a lists of semantic entities, including classes, properties and instances that match the search criteria. The resulting set, includes the URIs of the corresponding ontologies, so that they can also be retrieved and explored. This is especially useful in the context of creating annotations, since an overview of the complete ontology enables users to better decide if a certain semantic entity is suitable for making an annotation or not.

SWEET's architecture foresees an API for accessing ontology data (Figure 10.2). One implementation of this API is through using Watson for semantic search. In particular, the user can

select any keyword in the Web API HTML documentation, such as the input or output labels, and click on a button that completes the search. SWEET's implementation is not limited only to using Watson. In fact, other alternatives were tested, however, they returned only matching triples and no direct links to complete ontologies. Based on the gained experience with using SWEET, we found out that Watson and the option to directly retrieve and explore complete ontologies, in addition to the matching semantic entities, currently provides the best annotation search support. Still SWEET's API gives the possibility for integration with further semantic search providers, such as Sindice[6].

It is important to point out that currently SWEET relies strictly on the ranking of the results as provided by Watson. The only modification that is done on top of the results dataset is to include pagination. Therefore, future work in this respect will include ranking and possible filtering of the results. Still, the current implementation of SWEET, with the help of the integrated search support provided by Watson, significantly assists users in creating semantic Web API descriptions by taking away the extra effort necessary for finding suitable semantic entities.

The support in terms of assisting the user in annotating Web APIs can be taken one step further not only by enabling integrated search but also by directly suggesting suitable annotations. Such recommendations can be made for individual service parts or for the service as a whole. Some of the most commonly performed service tasks are based on the type of functionality that the service provides. The following section introduces an approach for automatically classifying Web APIs based on their HTML documentation and including this information as part of the service annotation.

10.3.2 Web API Classification Support

An important part of adding automation to the manual process of creating semantic descriptions of Web APIs is realised through supporting the functional classification of the APIs. In particular, determining the type of functionality of the service and its domain are one of the main tasks that need to be completed, when doing annotations recommendation. Knowing the type of a service eases not only the discovery and selection of services but also aids in determining the domain of the Web API and thus contributes towards selecting an appropriate domain ontology for making annotations. In this context, this section introduces two classification approaches – a three-step solution based on API HTML documentation in English and a multilingual classification approach that maps API documentation written in different languages to the same category.

The first classification solution foresees a three-step process and is an adaptation of a combination of previously developed and commonly applied classification algorithms [CD07]. Given the URL of the API documentation, the developed approach suggests a set of 5 ranked classes,

[6] `http://sindice.com/`

in accordance with the functionality types as defined by the ProgrammableWeb's directory[7]. The second classification solution takes into consideration the fact that currently providers publish the Web API documentation in any form and any language that they see fit and as a result not all descriptions are provided in English. Therefore, it is also important to be able to make classification recommendations for Web API descriptions that are given in a variety of different languages. For this purpose we present an approach that makes use of Cross-lingual Explicit Semantic Analysis [SC08] to classify and annotate APIs. As a result, once the semantic Web API descriptions are created with the help of SWEET, common service tasks can be performed based on the available metadata, such as the provided functionality type, and not solely based on the original, possibly non-English, textual documentation.

10.3.2.1 HTML-based Classification

The first classification solution is an adaptation of well-known previously developed classification algorithms [CD07]. Since it is based on a word-vector representation [Suz03] of the documents to be classified, it indirectly assumes that all documents are in the same language. Given the URL of an API documentation, the API's HTML page as well as a number of related pages (adjustable via a variable) are downloaded and processed. The processing includes pruning of the HTML, removing HTML tags, so that plain text can be extracted. Based on the retrieved text documentation, word-frequency weights and word stemming are performed [RNBY99]. As a result, the documents are converted in word-vector representations. In the final step, the so prepared input is processed with the k-nearest neighbour algorithm (KNN) [CD07] for classifying the API. A number of alternative classification algorithms were also used but KNN was chosen in the end because of delivering the best results. Algorithm 1 summaries the main steps of the classification solution.

Algorithm 1 HTML-based Classification

Require: webAPIDescriptionURL, n, classificationTaxonomy, backgroundCollection
htmlDocument ← download_html(webAPIDescriptionURL);
htmlRelatedDocuments ← download_realted_html(webAPIDescriptionURL);
prunedDocuments ← prune_html(htmlDocument, htmlRelatedDocuments);
wordVectorDocument ← convert_to_wordVector(prunedDocuments);
results ← compute_knn(wordVectorDocument, classificationTaxonomy, backgroundCollection);
return topN(results, n);

Our approach is very similar to kNN-based classification solutions, devoting extra effort to the preparation and processing of the Web API documentation. First, the pruning step includes removal of HTML-specific tags, as well as words that are very frequently used in the API context and carry no importance for determining the type of functionality of the service, such as 'API', 'parameter' and 'method'. Furthermore, individual words are weighted depending on

[7]The Programmable Web API directory `http://www.programmableweb.com/apis/directory`.

their positioning within the HTML. For example, words occurring in headings are assigned higher weights. As a result the kNN classification is applied on a word-vector representation of the original documentation that has lower level of noise-words and higher ranking for individual words that might be indicative for the type of API. The experiments conducted as part of the evaluation, which are described in the next chapter (see Section 11.4.2), actually point out that such pre-processing of the input is very beneficial.

Initially, the background collection (also referred to as the training set) was based only on API documentations and their classification as provided by ProgrammableWeb. However, this directory contains relatively few services per category for most categories[8], which is not optimal for training the classifier. Therefore, as an alternative training set, the Open Directory Project (ODP)[9] was used. The Open Directory Project is a manually created and managed directory of Web sites. It is maintained by volunteers who list web pages belonging to specific categories. 4,700,000 web pages are listed in almost 1,000,000 categories, and the data is freely and publicly available. The use of ODP was realised by creating a mapping between ProgrammableWeb's classification taxonomy and the one used by ODP. The developed classification recommendation approach was simplified and implemented within the scope of a master thesis [Ren10]. The classifier was realised with the help of Weka[10] and its implementation of the nearest neighbour algorithm. Following is a detailed description of the implementation of the classification approach.

Implementation. The classification approach is implemented in the form of a Service Classifier component. It is important to point out that since the approach is based on textual documentation, it can also easily be applied on WSDL-based services, in case that they have text-based descriptions in addition to the WSDL-files.

The Service Classifier architecture is shown on Figure 10.8. The workflow starts with the *Downloader* component. Its task is to fetch a webpage given its URL, return the content and report encountered fetch errors. The fetched pages are processed and cached for better performance to avoid fetching the same page multiple times.

When the main page describing the service is fetched and cached, it is sent to the *Parser-Sanitizer* component. The *Sanitizer* removes unnecessary elements from the HTML page, mainly JavaScript elements and flash objects (if any). The *Parser* analyses the different links of the page and requests related pages from the downloader. Based on experiments, fetching 5 of the sub-pages linked from the main page increases the accuracy by 10%. Therefore, retrieving related pages, in addition to the main service page, has an important effect on the accuracy of the component results.

[8] ProgrammableWeb API directory available at `http://www.programmableweb.com/apis/directory`

[9] `http://www.dmoz.org/`

[10] `http://www.cs.waikato.ac.nz/ml/weka/`

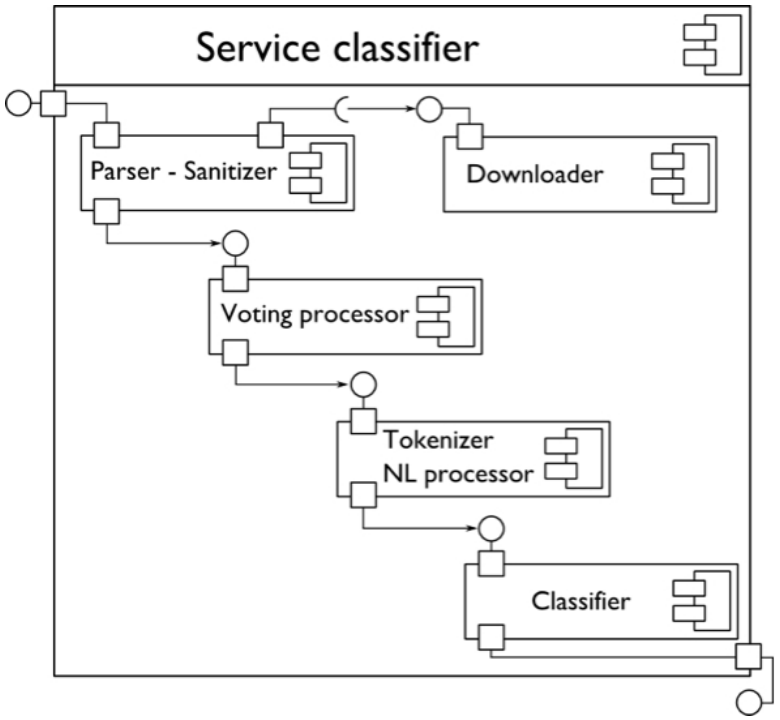


FIGURE 10.8: Service Classifier Component

Once a document and the set of related pages are collected and sanitised, they are given to the voting processor, which converts the different HTML pages to a weighted list of phrases and sentences. The type of HTML tags wrapping the text parts contribute to determining the weight of each phrase. The goal of this step is to convert a nested structure (a set of HTML pages) to a flatter list of 2-tuples, each one composed by a part of the document and its associated weight. The next step is to split the phrases and sentences into single words, and compute the weighted term frequency of each word. This list can potentially be very large and is thus pruned to accurately describe the set of documents: too-common and too-uncommon words are removed from the list.
The *Tokenizer* provides a list of words and their weighted frequencies. This list is given by the document storage component for persistent caching: the URLs and raw webpages are already stored, and the corresponding weighted term frequencies are attached to them. The frequencies are cached for performance reasons, because when the classifier processes a document, it needs to compare it to others, and the computation of the weighted list of term frequencies is an expensive task.

The communication between the individual components of the Service Classifier is visualised in Figure 10.9. In the final processing step, the *Classifier* component takes the tokenised text and determines the best matching category, out of a predefined list of categories, based on the data it has been trained with. This is done by calculating the distances between the document to classify, i.e., the Web API description, and the different documents as already classified in the background collection.

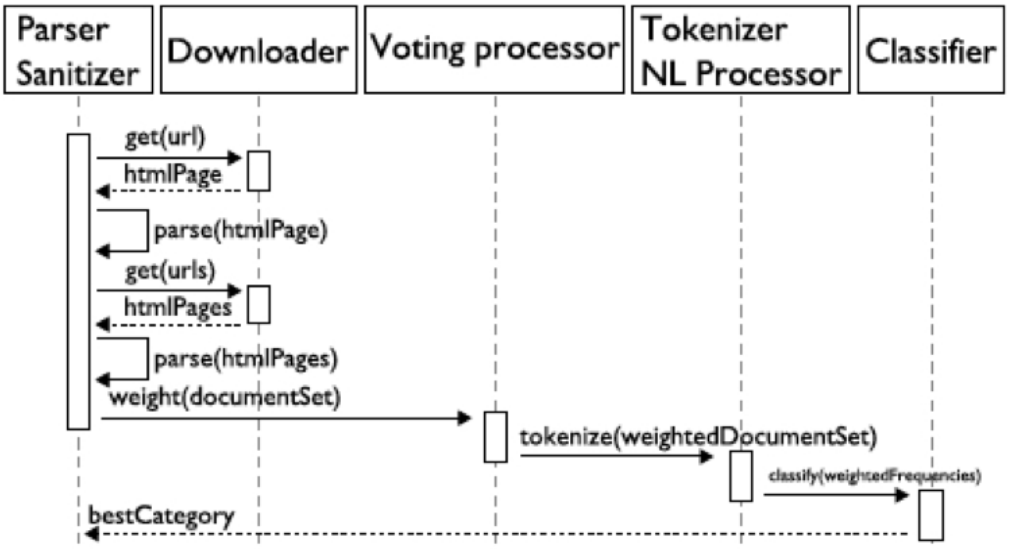


FIGURE 10.9: Service Classifier Component Communication

The classification step is potentially very computationally intensive – if a new document has to be compared to each existing document, this can be a problem especially in a large test collection. Therefore, a "typical document" is generated for each category – with one typical document per category, the number of operations will not grow with the number of classified documents. When a document is classified in a category, the "typical document" for this category is recomputed in order to take the new document into account.

```
1  public String classify(URL url);
2  public String[] classify(URL url, int matches);
```

LISTING 10.3: Service Classifier API

Once fully trained and setup, the classifier has a very simple API, as shown in Listing 10.3.

The Service Classifier does not have its own user interface, but uses the visualisation tool provided by Weka[11] in order to show the different steps involved in the classification process. As already mentioned, the service classification task requires some pre-processing work. Currently, all the steps preceding the actual classification are implemented in a standalone Java package that is able to crawl the web and output data in a format that can be used by the classifier. The tokenization is done by Lucene[12] and the pages are processed and sanitised using HtmlCleaner[13].

The classifier is implemented by Weka – the tokenizer outputs its data in a file that can be loaded by Weka to train and validate the classifier. The advantage of using Weka is that different classifiers and different options can be tested and compared before implementing the best one as a standalone component.

[11] `http://www.cs.waikato.ac.nz/ml/weka/`
[12] `http://lucene.apache.org/java/docs/index.html`
[13] `http://htmlcleaner.sourceforge.net`

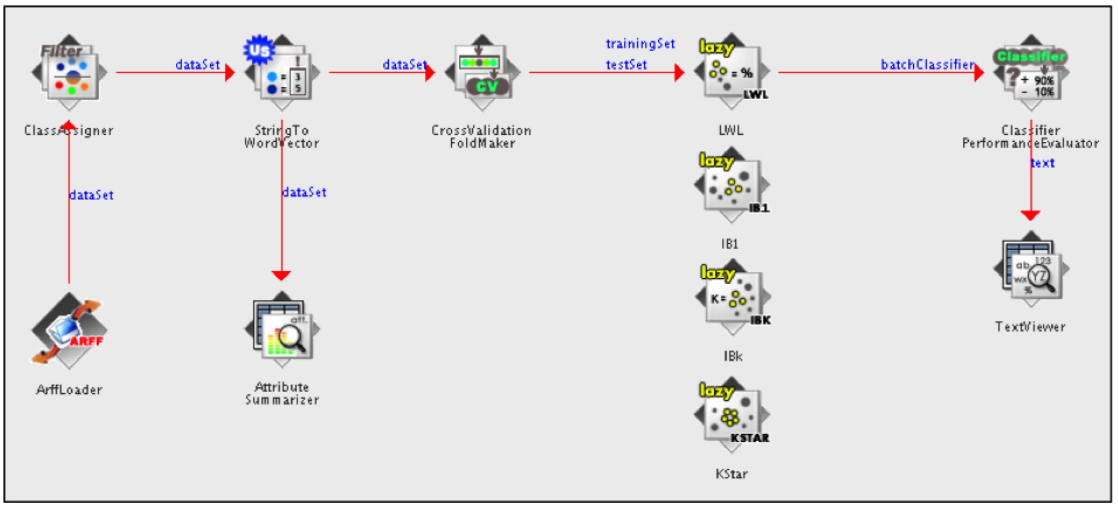


FIGURE 10.10: Classification Workflow

The experiment setup in Figure 10.10 shows how the classification is done:

1. Each Web API entry in ProgrammableWeb is crawled and passed through the tokenization component, which creates a file that is used as input to Weka.
2. The input data is pre-processed by Weka and the tokens are transformed to a "word vector" representation. In addition, each "word vector" is assigned a category, as already classified by ProgrammableWeb.
3. The classifier is setup to do 10-fold cross-validation of the classifier, i.e. the data set is split into ten separate folds and each fold is used to validate a classifier trained with the nine other folds. The advantage of cross-validation is that the risk of overfitting is much lower.
4. The training and validation sets are assigned to a classifier. The different classifiers can be swapped and the results of each run are stored.

As already mentioned, the training is done based on the background collection of already categorised data from the Open Directory Project[14] and ProgrammableWeb . The component is able to collect a list of links of ProgrammableWeb's API or an Open Directory Project data dump. Once the links are collected, they are processed by the *Downloader*, *Parser-Sanitizer* components and the resulting text is serialised into an ARFF[15] file. The ARFF file can be used as input to Weka, and is then loaded and processed as training data for the classifier.

The advantage of having such a setup is twofold. First, the training of the classifier can be done offline and independently of the actual classification during the Web API annotation process.

[14] `http://www.dmoz.org/about.html`
[15] `http://weka.wikispaces.com/Creating+an+ARFF+file`

Second, by using the data collected during each test run, the different classifiers can be compared and the influence of the individual parameter setting can be observed. As a result, the best performing one can be chosen for the actual Service Classifier component implementation. The so realised Service Classifier is available through a simple API, which has the service URL as input and a list of categories as output (Listing 10.3).

In summary, the here introduced classifier directly assists users in the process of adding semantic annotations to Web services. It is important to point out that the classifier is not restricted in particular to Web APIs, since it is based on analysing the textual content of the webpages describing the services, which can be both WSDL-based and RESTful. Given the URL to the documentation, it is able to suggest a category by using instance-based learning techniques. Experiments conducted using test data collected from ProgrammableWeb and training data from the Open Directory Project show that the service classifier component is able to determine the correct category among five different classes with more than 70% accuracy (see Chapter 11 for more details on evaluation).

10.3.2.2 Cross-Lingual Classification

The second classification solution employs a cross-lingual approach. In order to deal with the fact that not all Web API descriptions are in English and to be able to automatically determine the type of functionality that they provide, we present an approach that makes use of Cross-lingual Explicit Semantic Analysis [SC08]. Given a textual description and a background collection of APIs, the approach classifies the input API according to a predefined set of classes, independently of the language of the documentation. In particular, since Cross-lingual Explicit Semantic Analysis uses Wikipedia concepts to represent documents in a multilingual shared vector space, the approach is applicable to the majority of the languages.

Our approach towards Web API classification [MZKP11] is based on comparing the documentation of an API, which is to be classified, with a set of APIs, already classified according to a given taxonomy. The specific implementation is based on the ProgrammableWeb's taxonomy[16]. We refer to the set of pre-classified services as the *Background Collection*. In particular, we determine a number of representative service descriptions for each class in the taxonomy, while the actual classification process is not based on the textual documentation in the background collection but rather on the pre-computed Explicit Semantic Analysis (ESA) [GAS11] vector representations, where a word is represented as a column vector in the term frequency-inverse document frequency matrix (tf-idf matrix), having Wikipedia as a text corpus. A document is represented as a collection of the vectors representing its words. The ESA vectors are computed for each of the representative APIs, thus saving computation time at runtime.

[16] `http://www.programmableweb.com/apis/directory`

In addition, during the initial trial experiments, it was noticed that the Web API documentations use very limited vocabulary for describing the format of data and also for describing the behaviour of the Web API. For this reason, a *stop-word file* must be built to prevent the Explicit Semantic Analysis from focusing on the features of Web API descriptions that do not differentiate the services into classes. The stop-word list serves as an input for the pre-processing step of the Explicit Semantic Analysis (see Algorithm 2).

Algorithm 2 formally describes the proposed API classification approach. In particular, the devised method includes the following steps. First the language, in which the Web API description is written, is determined. This does not represent a challenge and can be done easily by comparing the word distribution of the Web API description to average word distributions of other languages, or using an existing solution in the form of a Web API[17]. Second, the Web API specific stop-words are removed and the Web API documentation is projected into the concept space given by the particular language version of Wikipedia. After that the vector is projected into the English Wikipedia concept space, to facilitate its comparison with our Web API background. The following step iterates over each document in the background collection and records its similarity with the previously determined vector of the input Web API documentation. Finally, for each category, the acquired similarity measures are summed up and divided by the total number of examples for the given category. This is done in order to derive a normalised similarity measure, which is not influenced by the number of representative services. The output is a list of categories, sorted according to their score.

Algorithm 2 Assigning Class Labels to a Web API Description

Require: webAPIDescription, backgroundCollection
Ensure: Scored class suggestions
  language ← recognize_language(webAPIDescription);
  esa_vector ← esa_analyze(language, webAPIDescription);
  esa_vector_en ← esa_map_vector(esa_vector, language, "en");
  category_score ← new Map();
  category_cnt ← new Map();
  for ⟨background_api_vector, category⟩ ∈ backgroundCollection **do**
    doc_score ← vector_similarity(esa_vector_en, background_api_vector);
    category_score[category] ← category_score[category] + doc_score;
    category_cnt[category] ← category_cnt[category] + 1;
  end for
  for category, score ∈ category_score **do**
    result[category] ← score / category_cnt[category];
  end for
  sort(result);
  return result

[17] `http://code.google.com/apis/language/translate/v1/using_rest_langdetect.html`

With Algorithm 2, Web API descriptions can be classified within the same category, independently of the language of the original documentation. Previous approaches base classification on word matches or word stemming/similarity and are, therefore, not applicable to a multi-lingual context.

The presented multi-lingual classification approach can be adapted in order to compute the central concepts of a given Web API documentation, based on the ESA vector representation. In particular, Algorithm 2 is extended with the computation of the similarity measure between the ESA vector of the API and the background collection. It is assumed that two APIs can be described with the same central concepts if their descriptions are semantically similar (i.e. their semantic relatedness measure is above some threshold). Our approach towards detecting the Central Concepts of a non-English Web API documentation is to find similar documentations in a repository of English-based APIs (in this approach serving as background collection), and re-use their central concepts (see Algorithm 3).

Algorithm 3 Determining the Central Concepts for a Web API Description

```
Require: webAPIDescription, backgroundCollection
  language ← recognize_language(webAPIDescription);
  esa_vector ← esa_analyze(language, webAPIDescription);
  esa_vector_en ← esa_map_vector(esa_vector, language, "en");
  for ⟨background_api_vector, central_concepts⟩ ∈ backgroundCollection do
    score ← cosine_similarity(esa_vector_en, background_api_vector);
    results[score] ← central_concepts;
  end for
  return max(results)
```

The benefits of extending the classification approach in order to determine the central concepts for an API description are multifold. First, they can be used directly as tags for the Web API. These tags can be employed to enhance search within directories or as complementary information presented to the user as part of the API description. However, with some further processing, the central concepts can serve as the basis for determining semantic annotations for separate service parts, such as inputs and outputs, or for extrapolating the domain of the service. In particular, it would be useful to input the computed words into Watson [dSM+08] or Sindice[18] and to use the results as suggestions for semantic entities suitable for annotating the API.

Implementation. The implementation of the cross-lingual Web API classification solution consists of three main parts. The first one is the *background builder*, which prepares the background collection for further classification. The second one implements the actual *cross-lingual classifier* based on the already introduced algorithm, while the third one performs *central concepts*

[18] `http://sindice.com`

detection. As background for the Explicit Semantic Analysis, we use different language versions of Wikipedia. The text analysis and its projection into ESA concepts space are done by our Java library, created by adapting the code from the Wikiprep ESA implementation[19].

The Web API background collection is built by getting APIs and categories from ProgrammableWeb. Five APIs are taken as an example for each category. Information about each API is saved to a database and after that the webpages describing each API are harvested. Subsequently, the HTML mark-up is removed and the text is normalised by removing stop-words and stemming. Then, the ESA vector is computed and stored in the database. Additionally, central concepts for each API in the background collection can be automatically determined by the AlchemyAPI[20]. Before putting the Web API documentation into the AlchemyAPI engine, we remove the API-specific stop-words.

Both, the classification and the central concept detection are implemented and operate similarly. In fact they differ only in the last step. They start with projecting the input API documentation into the corresponding language Wikipedia concept space. Then, the resulting language-specific ESA vector is mapped to an English ESA vector, using the concept mapping from Wikipedia. Afterwards, the ESA vector is compared with each API documentation ESA vector from the API background collection. The differing final step is as follows:

- In case of classification, the results are aggregated and the best categories are suggested as candidates.
- Concept detection does not summarise the results but rather suggests the central concepts of the first few most semantically similar Web APIs as concept suggestions.

The so computed results can be presented to the user as annotation suggestions, aiding the process of creating semantic Web API descriptions. In the case of the classification of the service functionality, the top 3 results, for example, can be automatically assigned to the API and the annotator would only need to validate them.

10.4 Summary

The benefits that come with a lightweight semantic service model cannot be fully used if the creation of the corresponding descriptions is too time- and effort-consuming. Therefore, this chapters introduces SWEET – a tool that supports users in annotating HTML API documentation and automatically generates RDF conforming to the MSM, which can directly be stored and

[19] `http://github.com/faraday/wikiprep-esa`
[20] `http://www.alchemyapi.com`

browsed in a service repository or be published as part of the provider's documentation. SWEET hides formalism complexity from the user and guides him/her through the process of marking the individual service properties and subsequently enhancing these with semantic metadata. The HTML documentation of the Web API is loaded in the tool, which provides functionalities for embedding microformat tags for syntactically structuring the description but also for linking to semantic entities. The resulting HTML can directly be published on the Web or can be used to extract RDF-based Web API descriptions. Without SWEET the user would have to revert to using a simple text editor and modify the HTML directly, which is a time-consuming and error-prone process.

Still, creating semantic Web API descriptions remains a challenging task, even with tool support in the form of SWEET. Therefore, we also introduce a number of solutions aiming towards supporting a semi-automated process of creating Web API descriptions. In particular, this includes integrated semantic entity search via Watson and via a customisable search API, as well as two classification approaches based on the type of functionally that the service provides. These solutions contribute towards automating common API annotation tasks, reducing the manual effort and, at the same time, lowering the adoption barrier.

Part IV

Evaluation and Conclusions

Chapter 11

Evaluation

In this chapter we present the evaluation of the Minimal Service Model, the Web API Grounding Model and the Web API Authentication Model. After having introduced each of the models in the previous chapters, here we determine how well they conform with the defined design requirements, what coverage they provide, given the heterogeneity of the Web API landscape, and what is the level of support in terms of enabling the automation of the invocation and authentication tasks. In particular, we revisit the characteristics that we specified and that the models need to fulfil, and use the data gathered by the two Web API surveys is order to determine, which common service properties are covered and which are not. We also provide an overview of the results of the evaluation of supporting tools and approaches for creating Web API descriptions. In particular we present the evaluation of SWEET and the two Web API classification approaches.

This chapter is structured as follows: Section 11.1 describes how MSM was evaluated, including the used criteria and a summary of the results. Section 11.2 covers the evaluation of the Web API Grounding Model in terms of conformity with the defined design requirements, the coverage that the model provides, and its level of support. Section 11.3 is structured in a similar way to Section 11.2 and presents the evaluation results of the Web API Authentication Model. Section 11.4.1 describes the evaluation of the supporting tools and approaches, which were developed in order to ease the creation of semantic Web API descriptions. This section includes evaluation of SWEET and of the two Web API classification approaches. We conclude the chapter with a summary given in Section 11.5.

11.1 Evaluation of the Core Service Model

In this section we describe the evaluation that was conducted on MSM. In particular, we evaluate the model in terms of its conformity to the requirements that we identified as prerequisites for

the developed solution. In addition, we also determine the coverage that MSM provides, given the current heterogeneity of the Web API landscape. We take into consideration established methods used for ontology evaluation [Vra10, PZ09]. Given the fact that MSM consist of only a few concepts and was developed following common ontology design principles, we do not follow all the evaluation steps as prescribed by [Vra10]. Still we consider the proper naming of elements, use of comments and labels, use of language tags, and checking of competency questions against results. Further evaluation methods, such as checking for superfluous blank nodes or inconsistency with rules, we omit because they simply do not apply in the context of MSM.

11.1.1 Requirements Coverage

During the requirements analysis discussed in Section 7.4, we identified the following requirements for developing MSM:

- R1: The model should be able to describe the majority (80% or more) of the APIs on the Web.
- R2: The core service model should take an operation-based view.
- R3: The core service model should be able to describe input and output data of the APIs' operations.

  R3.1: The core service model should be able to describe individual operation parameters but also the complete inputs and outputs as a whole.

  R3.2: The core service model should be able to describe optional and mandatory input parts.
- R4: The core service model should be able to describe the used HTTP method.
- R5: The core service model should be able to describe the endpoint URI.

The requirement related to the coverage of the model (R1) is a crucial one, not only for MSM but also for the Web API Grounding Model and the Web API Authentication Model, since without sufficient coverage, the models would not really be applicable. Furthermore, we tried to support as many of the Web API characteristics identified through the Web API surveys as possible, however, still keeping in mind that the model should enable the easy creation of annotations. Trying to find a balance between the level of detail and the level of complexity, we aimed for a coverage of at least 80%. This number was determined by identifying that about two thirds of the APIs have interfaces based on operations, and, in addition to that, we wanted to support at

least a half of the remaining APIs (resulting to a total of about 82%). As we demonstrate in the following section, we actually cover a larger percentage.

Overall, since currently there is no description model that is recognised to cover the majority of the existing APIs, based on capturing the characteristics provided in the documentation, it is very difficult to determine a baseline for comparing our results. In fact, none of the existing models discuss model coverage or provide corresponding measures and results.

Regarding the remaining requirements, MSM is based on the definition of a service that has a number of operations, which satisfies (R2). Furthermore, each operation has a *MessageContent*, which can be used to describe the input and the output (R3). The *MessageContent* can have individual message parts, for capturing input parameters, for example. As a result, both inputs and outputs can be described based on the complete data payload or based on its individual parts (R3.1). Furthermore, these individual parts can be defined as optional or mandatory (R3.2).

Finally, MSM is capable of describing the used HTTP method (R4) and the endpoint URI (R5) through two properties defined within the *rest:* namespace[1]. The reason why these properties are not directly included in MSM is that, despite the fact that they do represent two common Web API characteristics, they are relevant only for certain service tasks, specifically invocation. In contrast, discovery, composition, mediation, ranking and selection do not really rely on the provisioning of these details. Therefore, we internationally defined them within a different namespace, as an extension to MSM. In fact, these two elements play a key role in the Web API Grounding Model, also evaluated in the following sections. Following, is a discussion of the coverage provided by MSM, in the context of the heterogeneous Web API landscape.

11.1.2 Model Coverage

In this section we determine the coverage of MSM in terms of satisfying the competency questions and in terms of supporting the description of the majority (80% or more) of the APIs on the Web (R1 from the previous section).

In the process of designing the service model, we identified a number of competency questions related to the main service properties, which need to be captured, including their specific characteristics. We wanted to cover relevant information regarding

- What are the main service elements?
- What are the relationships between them?

[1] `http://purl.org/hRESTS`

MSM captures a service in terms of *Service*, *Operation*, *MessageContent* and *MessagePart*. The relationships between the service elements are refined with further questions:

- What are the operations of service X?
  MSM captures the operations of a service via the *hasOperation* property. A *Service* X *hasOperation* Y, where Y is of type *Operation*.
- What are the inputs and outputs of operation X?
  The inputs and outputs of an operation are captured via the *hasInput* and *hasOutput* properties, correspondingly. An *Operation* X *hasInput* Y, where Y is of type *MessageContent*. An *Operation* X *hasOutput* Y, where Y is of type *MessageContent*.
- What are the error messages of operation X?
  The error messages of an operation are captured via the *hasInputFault* and *hasOutputFault* properties. An *Operation* X *hasInputFault* Y, where Y is of type *MessageContent*. An *Operation* X *hasOutputFault* Y, where Y is of type *MessageContent*.
- What are the different parts of input X?
- What are the different parts of output X?
  Both the parts of the input and the output can be captured via the *MessagePart* class. Whether the *MessageContent* is an input or output is determined implicitly through the property that connects it to the operation (*hasInput* or *hasOutput*).
- What are the relationships between the parts of input/output X?
  MSM captures the relationships between parts of the input or output via the *hasPart* property. A *MessageContent* X *hasPart* Y, where Y is of type *MessagePart*.
- What are the different types of input/output parts?
  The different types of input and output parts are specified by defining two subproperties of the *hasPart* property (instead of defining subclasses of the *MessagePart* class). A *MessageContent* X *hasOptionalPart* Y, where Y is of type *MessagePart*. A *MessageContent* X *hasMandatoryPart* Y, where Y is of type *MessagePart*.

The operation endpoints and the HTTP method, which can be used to call it, are defined as part of the Web API Grounding Model, since they are especially relevant for invocation. These can be imported in MSM. We modelled this by reflecting on:

- What are the HTTP methods of operation X?
  The HTTP method of an operation is captured via the *hasMethod* property and *Method* class, defined in the Web API Grounding Model. An *Operation* X *hasMethod* Y, where Y is of type *Method*.

- What are the endpoints/addresses for operation X?
  The address of an operation is captured via the *hasAddress* property and *URITemplate* class, defined in the Web API Grounding Model. An *Operation* X *hasAddress* Y, where Y is of type *URITemplate*.

Therefore, we can conclude that MSM covers all the identified competency questions. We also evaluate MSM in terms of the general coverage that it provides. In particular, given the diversity of the Web API description forms and structures, as reflected in Chapter 6.3, it is important to evaluate MSM based on the coverage that it provides in terms of capturing API characteristics. In particular, we determine the overall percentage of coverage, depending on different API characteristics.

Characteristic	**% of APIs**	**Model Support by MSM**	**Model Part**
Type of Web API:			
Operation-Based Descr.	63.4	all	*msm:Operation*
Resource-Based Descr.	36.6	derived operation	*msm:Operation*
Input Details:			
Input and Output Parts	>55.2[1]	all	*msm:MessagePart*
No Input and Output Parts	<44.8[1]	all	*msm:MessageContent*
Optional Parameters	55.2	all	*msm:hasOptionalPart*
No Optional Parameters	44.8	all	*msm:hasPart*
Required Parameters	55.2	all	*msm:hasMandatoryPart*
No Required Parameters	44.8	all	*msm:hasPart*
Alternative Param. Values	63.9	-	*sawsdl:loweringSchemaMapping*
No Alternative Param. Values	36.1	-	*sawsdl:loweringSchemaMapping*
Default Parameter Values	51.7	-	*sawsdl:loweringSchemaMapping*
No Default Parameter Values	48.3	-	*sawsdl:loweringSchemaMapping*
Coded Parameter Values	34.9	-	*sawsdl:loweringSchemaMapping*
No Coded Parameter Values	65.1	-	*sawsdl:loweringSchemaMapping*
Boolean Parameters	33.7	-	*sawsdl:loweringSchemaMapping*
No Boolean Parameters	66.3	-	*sawsdl:loweringSchemaMapping*
State Parameter Datatype	27.9	-	*sawsdl:loweringSchemaMapping*
Do Not State Parameter Datatype	72.1	-	*sawsdl:loweringSchemaMapping* + manual testing
Output Formats:			
XML or JSON Output	>85.6	-	*sawsdl:liftingSchemaMapping*
Other Output Format	>14.4	-	*sawsdl:liftingSchemaMapping* + additional implementation
Invocation Details:			
Provide HTTP Method	59.3	-	Web API Grounding Model
Do Not Provide HTTP Method	40.7	-	Web API Grounding Model + manual testing
Provide Endpoint URI	89.5	-	Web API Grounding Model
Do Not Provide Endpoint URI	10.5	-	Web API Grounding Model + manual testing

TABLE 11.1: Coverage Provided by MSM

[1]Based on the use of optional parameters, used as an indicator for the need to have input parts.

Table 11.1 visualises a list of the characteristics used as a basis for the analysis, which are derived directly from the characteristics for capturing the current state of APIs on the Web. The table also gives the percentages of Web APIs[2], to which these criteria are applicable, whether and to what extent they are supported by MSM, and if they are supported, specifically by which part of the model.

As can be seen, MSM supports the description of all operation-based Web APIs, while about a third of the APIs require deriving the operation, by combing the particular HTTP method and resource (see Chapter 7 Section 7.5.3.4). We identify the number of Web APIs that require the definition of input parts, based on the number of APIs that used optional parameters. This is done because we have not collected specific data about what percentage of the APIs use more than two parameters and, therefore, might require the differentiation between the input as a whole and its individual elements. We describe the handling of alternative parameters, default parameters, coded parameters, boolean parameters and the input and output data transformations in general, in the following section while discussing the coverage provided by the Web API Grounding Model.

The percentage values for the different Web API characteristics are an indication of how frequently the particular characteristic is present in the documentation, i.e., how crucial it is that it is covered by the model. The higher the numbers, the more important it is that we provide support for it. Since we aim to cover the majority of the APIs, 80% or more, we can disregard any characteristic that is present in 20% of the cases or less. Overall, we cover all the listed characteristics (marked with 'all'), providing coverage of 100%. The only exception is the combination of resources and methods to derive operations, in the case of APIs that are resource-based.

It is important to point out that Table 11.1 does not provide a list of all Web API characteristics, but rather only the ones that were identified as relevant for MSM. Further, characteristics and their coverage by the models are given in Tables 11.3 and 11.5. In particular, we discuss the features relevant for invocation and authentication in the following sections.

In summary, the data clearly demonstrates that MSM captures all common Web API characteristics and the coverage that it provides is very high, given the diversity in the context of Web APIs. However, there are two issues that need to be mentioned. First, MSM does not support the description of resource-based services directly, but rather requires the definition of the corresponding operations in terms of combining the resource and the used HTTP method. Second, in the cases where the particular element is not described as part of the documentation, which is frequently true for the endpoint URI or the method, the corresponding MSM property can still be captured but would require some manual effort. However, in such situations MSM can be

[2]The numbers are based on the results of the second Web API survey. The percentages are used to indicate how important a certain characteristic is, based on how often it is present in current documentations. Since the results of the two survey are very similar (never more than 10% apart), we chose the data from the second survey, it being more up to date.

used as the basis for reducing underspecification, by serving as a guideline for providers, stating what is the minimum of description details that they need to include in the documentation. In the cases of missing details in the HTML documentation, tools such as SWEET (see Chapter 10.2) can be used to insert the missing properties.

11.2 Evaluation of the Web API Grounding Model

In this section we describe the evaluation results of the Web API Grounding Model. In particular, we conduct evaluation in terms of conformity with the defined design requirements, the coverage that the model provides, and the level of support, based on a number of Web APIs reflecting different invocation-relevant characteristics.

11.2.1 Requirements Coverage

We shortly revisit the model requirements for the Web API Grounding Model (Chapter 8) and discuss how well they have been fulfilled. Following is a summary of the requirements for developing the grounding model:

- R1: The HTTP method should be explicitly specified as part of the API description.
- R2: The description model should support parameterised URIs.
- R3: The description model should support the definition of an invocation address.
- R4: The relationship between the input parts and the HTTP requests should be specified.
- R5: The input data transformations should be defined.
- R6: Web API characteristics that directly determine how invocation is done should be captured.
- R7: The description model should support capturing the input as a whole, as well as its individual parts.
- R8: The output data transformations should be defined.
- R9: The output data transformations should be able to handle custom errors.

The design requirements for the Web API Grounding Model are derived directly based on the steps needed for creating, sending and processing HTTP messages, which are the basis for completing API invocation. Table 11.2 visualises the requirements and their realisation as part of

the model. In particular, the model captures the HTTP method (R1) and is based on defining the service and operation address (R3) in terms of URI templates that support the parameterisation of the URI (R2). The model also provides means for specifying data grounding via the *isGroundedIn* property, in particular defining whether the input values are transmitted as part of the HTTP body, header, or the URI (R4).

Requirement	**Covered by Model Part**
R1: HTTP method	*rest:hasMethod*
R2: Parameterised URIs	*rest:URITemplate*
R3: Invocation address	*rest:hasAddress*
R4: Input mapping to HTTP request	*rest:isGroundedIn*
R5: Input data transformations	*sawsdl:loweringSchemaMapping*
R6: Optional/required parameters and further	*msm:hasOptionalPart*/*msm:hasMandatoryPart*
R7: Input and input parts	*msm:MessageContent* and *msm:MessagePart*
R8: Output data transformations	*sawsdl:liftingSchemaMapping*
R9: Output data transformations for custom errors	*msm:hasOutputFault* + *sawsdl:liftingSchemaMapping*

TABLE 11.2: Fulfilment of the Design Requirements for the Web API Grounding Model

Lifting and lowering schema mappings (R5 and R8) can be associated with the inputs and outputs as a whole, i.e. *MessageContent*, but also with individual message parts (R7). In particular, we allow for fine-grain definition of the inputs and outputs (R7) that can have optional or mandatory parts (R6). In its original version, hRESTS expected a single lowering transformation that would apply to the whole input message, without distinguishing between different parameters of the URI, HTTP headers and the HTTP request message body. In our extension, we allow finer-grained (and thus more reusable) lowering transformations on individual message parts. Errors are handled via the definition of output faults and as part of the lifting transformation, which can be adjusted to deal with custom errors (R9).

11.2.2 Model Coverage

Similarly to MSM, we determine the coverage of the Web API Grounding Model in terms of satisfying the competency questions and in terms of covering the posed design requirements.

We identified a number of competency questions related to service elements, which need to be captured as part of the Web API Grounding Model in order to support invocation. In particular, we wanted to cover relevant information regarding:

- What are the main elements required for invocation?
- Are there relationships between them?

The Web API Grounding Model uses the core service model provided by MSM and captures invocation-relevant elements in terms of *URITemplate*, *Method*, *MediaType*. The relationships between the service elements are refined with further questions:

- What are the addresses for service X?
  The Web API Grounding Model captures the address of a service via the *hasAddress* property and the *URITemplate* class. A *Service* X *hasAddress* Y, where Y is of type *URITemplate*.

- What are the addresses for operation X?
  Similarly to the address of a service, the address of an operation is captured via the *hasAddress* property and the *URITemplate* class.

- What are the relationships between a service address and an operation address?
  The instance of the *URITemplate* assigned to the service, can be further specified of overwritten by the URI associated with the operation. This is not modelled explicitly as part of the Web API Grounding Model but is rather realised by using the URI template properties.

- What are the HTTP methods of operation X?
  The HTTP methods are captured via the *hasMethod* property and the *Method* class. An *Operation* X *hasMethod* Y, where Y is of type *Method*.

- What are the different parts of input/output X?
  The parts of the input and the output can be captured via the *MessagePart* class defined in MSM. Whether the *MessageContent* is an input or output is determined implicitly through the property that connects it to the operation (*hasInput* or *hasOutput*).

- What are the relationships between the input parts?
  The relationships between the input and its parts are captured in MSM with the help of the *hasOptionalPart*, *hasMandatoryPart* and *hasPart* properties. Mandatory and optional input parts have a direct influence on invocation (R6) and this is why we revisit them here.

- How is the input data transformation for input/input part X done?
  The input transformations are defined by using the SAWSDL property *loweringSchemaMapping*. A *MessageContent*/*MessagePart* has *loweringSchemaMapping*.

- What is the content type of input/input part X?
  The type of content that is produced by the lowering transformation is specified with the *producesContentType* property and the *MediaType* class.

- What are the relationships between the output parts?
  Similarly to the input, the output parts and their relationships are defined with the help of the properties defined in MSM.

- How is the output data transformation for output/output part X done?
  The output transformations are defined by using the SAWSDL property *liftingSchemaMapping*. A *MessageContent*/*MessagePart* has *liftingSchemaMapping*.

- What is the content type of output/output part X?
  The type of content that is expected by the lifting transformation is specified with the *acceptsContentType* property and the *MediaType* class.

- Which part of the HTTP message is used to send input/input part X?
  The Web API Grounding Model determines the part of the HTTP message used for transmitting the input via the *isGroundedIn* property. A *MessageContent*/*MessagePart* X *isGroundedIn* Y, where Y is http:Body, an instance of http:HeaderName, or a literal.

- What are the errors that can occur before the calling of operation X?
  Errors are specified with the help of properties defined in MSM. We revisit them here because of their relevance for invocation (R9). Errors that can occur before the calling of an operation are described via the *hasInputFault* property. An *Operation* X *hasInputFault* Y, where Y is of type *MessageContent*.

- What are the errors that can occur after the calling of operation X?
  Similarly to input faults, errors that can occur after the calling of an operation are described via the *hasOutputFault* property. An *Operation* X *hasOutputFault* Y, where Y is of type *MessageContent*.

Given the current heterogeneity of the Web API landscape, as reflected in Chapter 6, it is important to evaluate the Web API Grounding Model based on the coverage that it provides. In particular, we determine the overall percentage of coverage, depending on different API characteristics. Table 11.3 gives a list of the characteristics used as a basis for the analysis, which are derived directly from the characteristics for capturing the current state of APIs on the Web[3]. The table also provides the percentage of Web APIs, to which these criteria are applicable, whether and to what extent they are supported by the Web API Grounding Model, and if they are supported, specifically by which part of the model.

Table 11.3 is divided into five sections. The first three sections were already presented as part of determining the coverage of MSM (see Section 11.1) but are also relevant in the context of supporting invocation. Therefore, for the purposes of completeness, we have also included them here, since they are important also in terms of capturing all the API properties that are required for performing the invocation task.

[3]Similarly, to determining the coverage of MSM, the numbers are based on the results of the second Web API survey.

Characteristic	**% of APIs**	**Model Support by Grounding Model**	**Model Part**
Type of Web API:			
Operation-Based Descr.	63.4	-	*msm:Operation*
Resource-Based Descr.	36.6	-	*msm:Operation*
Input Details:			
Input and Output Parts	>55.2[1]	-	*msm:MessagePart*
No Input and Output Parts	<44.8[1]	-	*msm:MessageContent*
Optional Parameters	55.2	-	*msm:hasOptionalPart*
No Optional Parameters	44.8	-	*msm:hasPart*
Required Parameters	55.2	-	*msm:hasMandatoryPart*
No Required Parameters	44.8	-	*msm:hasPart*
Alternative Param. Values	63.9	-	*sawsdl:loweringSchemaMapping*
No Alternative Param. Values	36.1	-	*sawsdl:loweringSchemaMapping*
Default Parameter Values	51.7	-	*sawsdl:loweringSchemaMapping*
No Default Parameter Values	48.3	-	*sawsdl:loweringSchemaMapping*
Coded Parameter Values	34.9	-	*sawsdl:loweringSchemaMapping*
No Coded Parameter Values	65.1	-	*sawsdl:loweringSchemaMapping*
Boolean Parameters	33.7	-	*sawsdl:loweringSchemaMapping*
No Boolean Parameters	66.3	-	*sawsdl:loweringSchemaMapping*
State Parameter Datatype	27.9	-	*sawsdl:loweringSchemaMapping*
Do Not State Parameter Datatype	72.1	-	*sawsdl:loweringSchemaMapping* + manual testing
Output Formats:			
XML or JSON Output	>85.6	-	*sawsdl:liftingSchemaMapping*
Other Output Format	>14.4	-	*sawsdl:liftingSchemaMapping* + additional implementation
Invocation Details:			
Provide HTTP Method	59.3	all	*rest:hasMethod* + *rest:Method*
Do Not Provide HTTP Method	40.7	-	manual testing
Provide Endpoint URI	89.5	all	*rest:hasAddress* + *rest:URITemplate*
Do Not Provide Endpoint URI	10.5	-	manual testing
Invoc. URI with URI templates	70.3	all	*rest:hasAddress* + *rest:URITemplate*
Invoc. URI without URI templates	29.7	all	*rest:hasAddress* + *rest:URITemplate*
Invoc. URI uses query parameters	77.3	all	*rest:URITemplate* + *rest:isGroundedIn*
Invoc. URI without query parameters	32.7	all	*rest:hasAddress* + *rest:URITemplate*
Input values in URI	78.5	all	*rest:URITemplate* + *rest:isGroundedIn*
Input values in HTTP Header	0.6	all	*rest:isGroundedIn*
Input values in HTTP Body	18.6	all	*rest:isGroundedIn*
Input values - mixed	1.7	all	*rest:URITemplate* + *rest:isGroundedIn*
Require Construction of HTTP Request	>29.2[2]	all	Complete Grounding Model + through *rest:isGroundedIn*
Invocable via URL	~ 70[3]	all	*rest:URITemplate* + *rest:isGroundedIn*
Description of Errors:			
APIs without description of Errors	45.4	-	manual testing
APIs with description of Errors	54.6	-	*msm:hasInput/OutputFault* + *msm:MessageContent*
APIs with Custom Errors	29.1	-	*sawsdl:liftingSchemaMapping* + additional implementation
APIs with standard HTTP Errors	25.5	-	*sawsdl:liftingSchemaMapping*

TABLE 11.3: Coverage Provided by the Web API Grounding Model

[1]Based on the use of optional parameters, used as an indicator for the need to have input parts.

[2]Based on the percentage of APIs that require the construction of the HTTP Header or Body, for transmitting input or authentication credentials.

[3]Based on the percentage of APIs that do not require the construction of the HTTP Header or Body.

This includes in particular, handing the different types of parameters, including alternative parameters (1, 2, or 3), default parameters, coded parameters ('en' instead of 'english') or boolean parameters ('true' or 'false', 'yes' or 'no'). In general, the proper transformation of the data and the determining of the specific values are handled as part of the data lowering, thus enabling the actual processing on the level of semantic data. However, the practical implementation of the particular mappings might require more than a simple script that converts the input from one format (e.g. RDF) into another (e.g. String values). This is for instance true of encoded values, where the determining of the proper format of the input is based on a list of all possible abbreviations or codes for the values (for example, all country codes or all language codes). The proper processing in this case would require additional implementation efforts. However, such more complex cases can be solved by sharing the data transformations with potential client application developers, so that they can reuse the solution. On the level of providing coverage through the Web API Grounding Model, input and output data transformations are actually handled as part of the SAWSDL lifting and lowering schema mappings.

In the remaining two sections, we cover characteristics that are especially relevant for creating and processing the HTTP requests and responses. As can be seen, the invocation details are all supported, expect in the cases where the information is simply not provided as part of the documentation (for example, APIs that do not provide the HTTP method). In this case, the processing of the documentation would require additional effort in the form of test invocations. As can be seen, about one third of the APIs require the construction of the complete HTTP request, while about 70% of the APIs can be invoked through parameterised URLs. The unified handling of both types of invocation is only possible through the definition of the service and operation addresses in terms of URI templates, where the parameter can be assigned the appropriate input value, and through the use of the *isGroundedIn* property that specifies how the input is transmitted. The descriptions of errors are captured as part of MSM, where custom errors might present a challenge since each possible error occurrence needs to be captured as part of the lifting transformation (instead of relying on the use of standard HTTP errors).

We aim to provide wide coverage – supporting at least 80% of existing APIs. As can be seen in the table, the coverage provided by the Web API Grounding Model is indeed very high (hindered to a certain extent by missing information). Still, it needs to be pointed out that the lifting and lowering data transformations need to be provided in order to enable the automated completion of the invocation process. Depending on the used parameters and errors, these might require

additional implementation work that can be relieved to a certain extent by sharing and reusing data processing solutions.

The percentage values per API characteristic can be used to indicate how important a certain characteristics is, based on how often it is present in documentations. Therefore, in theory, we could omit characteristics with lower significance, such as for example, boolean parameters.

As previously pointed out, Web APIs are commonly underspecified, as is the case with 40% of the APIs that do not state the HTTP method or 72% that do not give the datatypes of the parameters. In this context, the grounding model can be seen as a way for counteracting underspecification, since it states all the service properties that need to be part of the description in order to guarantee the invocability of the API, thus encouraging providers to include this information.

Overall, based on the evaluation, we can conclude that the Web API Grounding Model appropriately facilitates the description and use of heterogeneous services by sharing the semantics of services through formal machine-processable descriptions as well as using a common syntax for representing these descriptions and the exchanged data.

11.2.3 Suitability for Purpose

Finally, based on a number of examples, we show that the model is suitable for annotating a wide range of Web API description types and forms, thus providing support for the majority of the APIs on the Web. In particular, we evaluate the Web API Grounding Model by annotating actual Web APIs and testing their invocability. The chosen APIs cover a range of different description forms and characteristics, in order to determine the variety of APIs that are practically invocable by OmniVoke.

Table 11.4 shows some of our test API examples[4] that represent different types of APIs, each posing individual requirements on the invocation model. As can be seen, we evaluated the coverage of each requirement multiple times and all resulting semantic descriptions were invocable by the invocation engine. We also tried to cover all the possible different alternatives within a requirement. One alternative that we did not cover was the passing of input parameters via the HTTP header, which occurs in less then 1% of the cases. Similarly, we did not cover HTTP DELETE. This HTTP method can easily be modelled by the Web API Grounding Model, but it is unfortunately still not supported by OmniVoke.

[4] All examples are available at `http://purl.org/hRESTS/`

Web API	R1: HTTP Method	R2: Parameterised URI	R3: Invocation Address	R4: Input Grounding
Nestoria searchListings	GET[1]	Yes	Yes	in URI parameter
Ribbit sendSMS	POST	Yes	Yes	in HTTP Body
Ribbit uploadMedia	POST	Yes	Yes	in HTTP Body
GeoNames getCountryCode	GET[1]	Yes	Yes	in URI parameter
Last.fm getArtistInfo	GET[1]	Yes	Yes	in URI parameter

	R5: Input Transformation	R6: Opt. Param	R7: Msg. Parts	R8: Output Transformation	R9: Custom Errors
Nestoria searchListings	Yes	Yes	Yes	Yes	N/A
Ribbit sendSMS	Yes	Yes	Yes	Yes	N/A
Ribbit uploadMedia	Yes	Yes	Yes	Yes	N/A
GeoNames getCountryCode	Yes	Yes	Yes	Yes	Yes
Last.fm getArtistInfo	Yes	Yes	Yes	Yes	Yes

TABLE 11.4: Test Web API Invocation Descriptions

'Yes' indicates that a certain requirement is covered by the example.

[1]The HTTP method was not explicitly stated as part of the documentation and we had to do a test invocation in order to be sure that it is GET.

In the case of the telecommunication Ribbit service, the operation needed to be derived based on combining the resource with the corresponding method, in order to create an operation-based semantic description. This was necessary since the particular interface is based on resources, instead of operations. In the cases where some details were missing, such as the HTTP method or the input parameter types, these needed to be determined via trial and error, and added to the semantic description. This was the case with the Nestoria, GeoNames and Last.fm APIs, which are affected by some underspecification. Still, we can say that we have evaluated our approach by creating annotations that cover all of the defined requirements and have successfully used them to perform invocation.

11.3 Evaluation of the Web API Authentication Model

The Web API Authentication Model is evaluated based on the same criteria as the Web API Grounding Model – conformity to competency questions and design requirements, coverage of the model, and suitability for purpose.

11.3.1 Requirements Coverage

Similarly to MSM and the Web API Grounding model, we evaluate the Web API Authentication (WAA) model in terms of satisfying the competency questions and in terms of covering all the requirements, which were identified while designing the model.

The process of defining WAA was guided by a number of competency questions. We determine how well they are actually covered by the resulting model. Relevant information to identifying when authentication is required and the information that is needed is:

- Does the service require authentication?
  The WAA captures whether a service requires authentication or not via the *requiresAuthentication* property. A *msm:Service* X *requiresAuthentication* Y, where Y is of type *ServiceAuthentication*. The *ServiceAuthentication* class has three instances – *All*, *Some*, and *None* that are used to indicate if authentication is required for all operations, only for some or for none of the operations.

- Which operations require authentication?
  Operations that required authentication are described via the *hasAuthenticationMechanism* property.

- What kind of authentication is used?
  The particular kind of authentication that is used is captured via the *AuthenticationMechanism* class. A *msm:Service/msm:Operation* X *hasAuthenticationMechanism* Y, where Y is of type *AuthenticationMechanism*.

- What is the required information to complete the authentication?
  Authentication has three main characteristics, including the credentials, the authentication protocol, and the way of sending the authentication information. These are captured with the help of the *Credentials* class, the *AuthenticationMechanism* class, and the *wayOfSendingInformation* property.

We can refine the information necessary for supporting a particular authentication mechanism by determining:

- What are the required credentials?
  WAA captures credentials via the *Credentials* class. An *AuthenticationMechanism* X *hasInputCredentials* Y, where Y is of type *Credentials*. The *Credentials* can be further refined by using seven subclasses – *APIKey*, *Username*, *Password*, *OAuthConsumerKey*, *OAuthConsumerSecret*, *OAuthToken*, and *OAuthTokenSecret*.

- What is the used authentication protocol?
  The authentication protocol is captured with the help of the *AuthenticationMechanism* class. A *msm:Service/msm:Operation* X *hasAuthenticationMechanism* Y, where Y is of type *AuthenticationMechanism*. The *AuthenticationMechanism* can be further refined by using six subclasses – *HTTPBasic*, *HTTPDigest*, *OAuth*, *WebAPIOperation*, *SessionBased*, and *Direct*.

- How is the authentication information transmitted?
  The way of transmitting the authentication information is captured via the *wayOfSending-Information* property. The way that credentials are sent as part of the HTTP messages is described with the help of *isGroundedIn*, *hasValue*, and *hasName* properties.

We shortly revisit the model requirements for the Web API Authentication Model (Chapter 9) and discuss how well they have been fulfilled. Following is a summary of the requirements for developing the grounding model:

- R1: The used credentials should be specified.
- R2: The used authentication protocol should be specified.
- R3: The way of sending the authentication information should be specified.
- R4: The most commonly used authentication approaches should be covered.
- R5: The most commonly used credentials should be covered.
- R6: The most commonly used ways of sending the authentication information should be covered.

The design requirements for the Web API Authentication Model are derived directly based on the analysis of currently existing authentication approaches and the data collected on commonly used credentials and protocols. The model captures the authentication credentials through the *waa:Credentials* class (R1) and also provides subclasses for the most commonly used credential types (R5) (*waa:APIKey*, *waa:Username*, *waa:Password*, *waa:OAuthConsumerKey*, *waa:OAuthConsumerSecret*, *waa:OAuthToken*, *waa:OAuthTokenSecret*). Similarly, it defines the *waa:AuthenticationMechanism* class for describing the underlying protocols (R2) and individual subclasses for the frequently used ones (R4) (*waa:HTTPBasic*, *waa:HTTPDigest*, *waa:OAuth*, *waa:WebAPIOperation*, *waa:SessionBased*, *waa:Direct*). Finally, WAA also includes a property for the way for sending authentication information (R3) and takes into consideration the most commonly used ways of sending authentication credentials by including the *waa:isGroundedIn* property for specifying credentials, their values, and the way they are transmitted (R6).

11.3.2 Model Coverage

Table 11.5[5] visualises the most common authentication approaches, the corresponding percentage distributions, to what extent they are supported by WAA and by which specific parts. Here

[5]The numbers are based on the results of the second Web API survey.

again, the percentages are an indication of how relevant a certain approach is, based on its frequency of use.

Characteristic	**% of APIs**	**Model Support by WAA**	**Model Part**
Authentication Details:			
Require Authentication	79.7	all	Web API Authentication Model
No Authentication	20.3	N/A	N/A
Required Authentication:			
Authentication for All Operations	85.5	all	*waa:requiresAuthentication* + <http://purl.org/waa#All>
Auth. Only for Data Modification or Some Operations	4.6	all	*waa:requiresAuthentication* + <http://purl.org/waa#Some>
Offer Alternative Authentication Mechanisms	20.3	all	*waa:hasAuthenticationMechanism* + *waa:AuthenticationMechanism*
Authentication Mechanisms:			
API Key	30.8	all	*waa:Direct* + *waa:APIKey*
HTTP Basic	21.5	all	*waa:HTTPBasic* + *waa:Username* + *waa:Password*
Username and Password	7.5	all	*waa:Direct* + *waa:Username* + *waa:Password*
OAuth	12.8	all	*waa:OAuth* + *waa:OAuthConsumerKey* + *waa:OAuthConsumerSecret* + *waa:OAuthToken* + *waa:OAuthTokenSecret*
Web API Operation	4.6	all	*waa:WebAPIOperation*
HTTP Digest	5.8	all	*waa:HTTPDigest* + *waa:Username* + *waa:Password*
API Key in Combination with Other Credentials	11.6	all	*waa:Direct* + *waa:APIKey* + credentials
Session Based	1.7	all	*waa:SessionBased*
Other	0.6	-	-
Way of Transmitting Credentials:			
URI	69.3	all	*waa:isGroundedIn*
HTTP Header	23.4	all	*waa:isGroundedIn*
HTTP Body	5.8	all	*waa:isGroundedIn*

TABLE 11.5: Coverage provided by the Web Authentication Model

The Web API Authentication Model covers APIs, which use an API Key, via the *waa:Direct* authentication mechanism and *waa:APIKey*. In addition, the HTTP Basic and Digest protocols are covered as well. Authentication only via credentials, which is described as *waa:Direct*, is covered to the most part via predefined classes for credentials. In the case of 'API Key in Combination with Other Credentials', if the needed credentials are not already defined as subclasses, the *waa:Credentials* class can be used directly. Finally, OAuth, authentication done over an especially dedicated operation, as well as session-based authentication are supported via corresponding authentication mechanisms.

In summary, the Web API Authentication Model does not cover directly only the 1% of 'Other' authentication mechanisms. Therefore, based on the collected data, we can conclude that WAA supports close to a 100% of the APIs that require some form of authentication, thus providing

the aimed for wide coverage. It is important, to keep the high level of support by updating and extending the model with the newest and most relevant authentication approaches.

11.3.3 Suitability for Purpose

Finally, we evaluate the Web API Authentication Model by annotating actual Web APIs and testing their invocability. The chosen APIs cover a range of different authentication approaches, in order to determine the range of APIs that are practically invocable by OmniVoke. In particular, we cover each of the identified six authentication mechanisms at least once and also include APIs requiring a variety of credentials.

Web API	**Direct**	**HTTPBasic**	**HTTPDigest**	**OAuth**	**APIOp**	**Session**
Eventful	Yes	Yes	Yes	-	-	-
Flickr	-	-	Yes	-	-	-
GeoNames getCountryCode	Yes	-	-	-	-	-
Last.fm getArtistInfo	Yes	-	-	-	-	-
ClearForest Semantic Web Services1	Yes	-	-	-	-	-
Basecamp	-	Yes	-	-	-	-
Daylife	-	-	Yes	-	-	-
Adobe Share	-	-	-	-	-	Yes
Docstoc	-	-	-	-	Yes	-
Yelp	-	-	-	Yes	-	-

TABLE 11.6: Test Web API Authentication Descriptions

'Yes' indicates that a certain requirement is covered by the example.

Table 11.6 shows some of our test API examples that represent different types of APIs, each posing individual requirements on the authentication model[6]. As can be seen, we covered all authentication mechanisms for the Web API Authentication Model. In addition, we test different credentials that can be used for one and the same authentication mechanism, such as API key or username and password for *Direct* authentication.

The only APIs that presented somewhat of a challenge were the ones based on authentication through individual API operations[7], since this implies that an additional request needs to be made before the API can actually be called. In summary, we have evaluated our approach by creating annotations that cover all of the authentication mechanisms and a set of credentials, and have successfully used them to perform authentication as part of the invocation process.

[6] All examples are available at `http://purl.org/waa/`

[7] For example, Docstoc – `http://platform.docstoc.com`

11.4 Evaluation of Supporting Tools and Approaches

In this section we present the evaluation results of SWEET and of the two Web API classification approaches.

11.4.1 Evaluation of SWEET

When it comes to evaluating SWEET, there were a number of different approaches used, not focusing strictly on gathering feedback about the provided user interface but also on evaluating the available functionalities. In particular, the benefits of SWEET were determined in four main ways. First, the tool was used in a number of project use cases in order to annotate Web APIs from particular domains (music domain and geocoding domain). Second, it was used as part of hands-on sessions and tutorials, where direct feedback was gathered. Third, a questionary was used in order to determine the tool's weaknesses and strengths and finally, all user actions were anonymously recorded, in order to determine, which are the most commonly performed annotation tasks, which activities take the longest and which are the areas of possible improvement. Each of these evaluation strategies provides facets of different insights about SWEET and points out issues that need more attention or better handling.

Project Use Cases. The Web application version of SWEET was developed as part of a suite of tools targeted as supporting tasks along the lifecycle of services, within the scope of the SOA4All[8] European project (FP7 - 215219). The tool was used in one of the three project use cases in the context of telecommunication services. It was used in two scenarios – creating a mashup for event booking and search, and developing a mobile app for web-based messaging. In this context SWEET enabled the creation of semantic Web API descriptions, which were subsequently integrated as part of service compositions. The development of SWEET within the scope of the project is documented in a number of deliverables[9], the main ones being the ones on the service provisioning platform, which encompasses components that support the creation of service descriptions, including Web APIs and traditional Web services.

The work on providing a tool that supports users in creating semantic service description was done in parallel to the efforts towards defining a formalism for the semantic description of Web APIs. More importantly, SWEET's development was directly influenced by the feedback and requirements resulting from the use cases, which provided grounds for testing and improving the practical applicability of the tool. In addition to creating the descriptions used as part of the

[8] http://cordis.europa.eu/project/rcn/85536_en.html

[9] http://cordis.europa.eu/docs/projects/cnect/9/215219/080/deliverables/D2-1-3-SERVICE-PROVISIONING-PLATFORM-FIRST-PROTOTYPE.pdf, http://cordis.europa.eu/docs/projects/cnect/9/215219/080/deliverables/001-D214SERVICEPROVISIONINGPLATFORM2NDPROTOTYPE.pdf

composition scenarios, SWEET was also one of the main tools presented as part of a hands-on session demonstrating results of the project. Therefore, the work done within the scope of the project provided a good basis for evaluating the suitability of the tool's functionalities for supporting actual use cases and contributing to solving existing challenges in the context of particular industry domains, such as telecommunications.

Training Sessions and Tutorials. SWEET was used in a number of tutorials and training sessions, demonstrating the practical applicability and benefits of the semantic Web API models. In particular, the first version of the Web application tool was part of hands-on sessions in two summer schools (The Summer School on Service and Software Architectures, Infrastructures and Engineering 2009 (SSAIE Summer School) – about 30 participants, and the 1st Karlsruhe Summer School on Service Research – about 20 participants). These sessions demonstrated the usability of the tool, when it comes to creating semantic descriptions of Web APIs, but also provided valuable feedback about the user experience and pointed out some deficiencies. The tutorials were used to evaluate the current prototype and determine necessary extensions and further required functionalities. The main comments were related to the need for more flexibility in making the annotations, in particular when it comes to correcting mistakes. Therefore, functionalities such as deletion, renaming and updating were in the focus of the implementation work done as part of the following prototypes of SWEET (R1 for the extended version of SWEET).

This led to the development of the extended version of SWEET, which was used in the The Summer School on Service and Software Architectures, Infrastructures and Engineering 2010 (SSAIE Summer School) in a tutorial on Linked Services with about 30 participants, and subsequently in a tutorial on Automating the Use of Web APIs through Lightweight Semantics at the International Conference on Web Engineering (ICWE 2011) with about 15 participants. Overall, the gathered feedback was quite positive, since SWEET enables the annotation of Web APIs, which would otherwise have to be done with the help of applications such as a simple text editor. Therefore, the majority of the tutorial participants found the tool useful. There were also some useful suggestions when it comes to improving the user interface. In order to be able to properly gather feedback on SWEET, a short survey was conducted after each training session. The survey layout as well as the main input is described in more detail in the following section.

User Survey. The gathering of feedback and recommendations, and implementing these in subsequent versions of the tool, is an important part of improving SWEET based on user requirements. For this purpose, a simple survey was designed, which was distributed to the training session participants. The first section gathers some information about the background of the users. This includes questions on previous experience with Web services, semantics and making annotations. The second section includes specific questions regarding the functionality and

support provided by SWEET, including estimating the level of difficulty of the individual annotation tasks, the effort required for creating the semantic description and the intuitive use of the tool. The final section includes a set of questions that can be answered with free text. These are targeted at collecting recommendations and making improvement suggestions.

Even though during the past three years SWEET was used in more than six training sessions, the survey was completely only by a total of 17 participants. The majority of the participants were PhD students with some previous experience in services and knowledge of semantics. Here we summarise some of the main findings of the survey. First, all participants said that having a tool such as SWEET is very helpful in doing annotations, since without it this has to be done is a simple text editor directly editing the HTML. The majority of the participants said that the most useful functionality is the integrated search for semantic entities, since without previous knowledge in a particular domain finding suitable annotations is a time-consuming task. Finally, some participants found it difficult to identify the different service parts that needed to be annotated. This points out that some initial training is required before SWEET can be successfully used.

Overall, the task of annotating an API was not perceived as difficult and neither was the effort required for completing it. Regarding the general comments and suggestions, one participant said that it would be useful to implement the annotation process in a step-by-step manner, so that, for example, no semantic model references can be added before the insertion of hRESTS tags. This would provide for a more controlled and guided use of SWEET in creating semantic Web API descriptions. All the comments were taken into consideration in each of the subsequent versions of the tool.

Annotation Action Logging. SWEET was also evaluated implicitly by anonymously logging the individual annotation steps completed by the user. This was done by recording all the actions that are performed and storing them in a triplestore. Listing 11.1 shows the simple schema used for the *Logger* repository. Each *LogEnty* has a date-time stamp, a session id, and a number of actions. An *Action* has a set of subclasses that determine the specific task that was performed. All action subclasses start with the *Item* prefix. For example, an *ItemSave* instance is created when the user clicks on the Save button and stores the created annotation locally, while *ItemSaveToRepository* is used when the semantic description is posted directly to iServe. The benefit of logging the user actions is twofold. Not only do the logs deliver important details about the time needed to complete each annotation task but they can also be used for debugging purposes.

Since we do not know the user identity, we based the logging on the browser session. Therefore, all the action items belonging to one session are considered as done by the same user, as part of one annotation process.

```
1   s:LogEntry
2   s:Action
3   s:ItemSave
4   s:ItemExport
5   s:ItemSaveToRepository
6   s:ItemCreation
7   s:ItemSearchWatson
8   s:ItemHTMLAnnotation
9   s:ItemSemanticAnnotation
10  s:ItemCallProxy
11  s:ItemDeleteSemAnnotation
12  s:ItemDeleteHTMLAnnotation
```

LISTING 11.1: Actions Logging Schema

The goal of collecting this data is to be able to determine, which are the most commonly performed actions, how long does an annotation process take and which actions takes the longest. Therefore, if we are able to identify frequent and time-consuming tasks, we can focus on enhancing SWEET's functionality for supporting these particular tasks. Since most of the data was collected during the training sessions, all the tasks from those events had the same durations. Therefore, the identification of individual outliers was difficult. In some of the cases the session was lost and a new one was automatically created. We use the session to group the performed annotation tasks done by one user, therefore, tasks belonging to the same annotation log were recorded in two individual ones instead. Overall, we found out that the most commonly performed task was *ItemHTMLAnnotation*, while the deletion actions were recorded the fewest times, probably because users made corrections instead of deleting the particular annotation. This implies that the focus of improving SWEET's functionalities should be on supporting the creation of annotations, in particular, of hRESTS and semantic annotations. Therefore, a simple and intuitive user interface is very important in this context.

Tool Uptake. In order to be able to follow the uptake and popularity of use of SWEET, we registered it with Google Analytics[10]. Google Analytics collects data about the number of site visits, geographical distribution of the visitors, number of unique visitors, used browser, etc., and provides different overviews and chart summaries as means for exploring the aggregated data. SWEET, both the tool website (`http://sweet.kmi.open.ac.uk`), as well as the demo (`http://sweetdemo.kmi.open.ac.uk`) were registered in mid-July 2009 with the launch of the first Web application version. The results presented here are for the actual deployment of the tool and demonstrate how often SWEET was used.

Figure 11.1 shows the monthly distribution of the number of visitors. As can be seen, the two peaks are during October/November 2009 and the summer of 2010 when most of the training sessions, as part of summer schools, took place. Even though the overall number of visitors is not that high (590), it is important to point out that about one third of those are returning visitors, who have used SWEET a number of times, accounting for a total of 411 unique visitors.

[10] `http://www.google.com/analytics/`

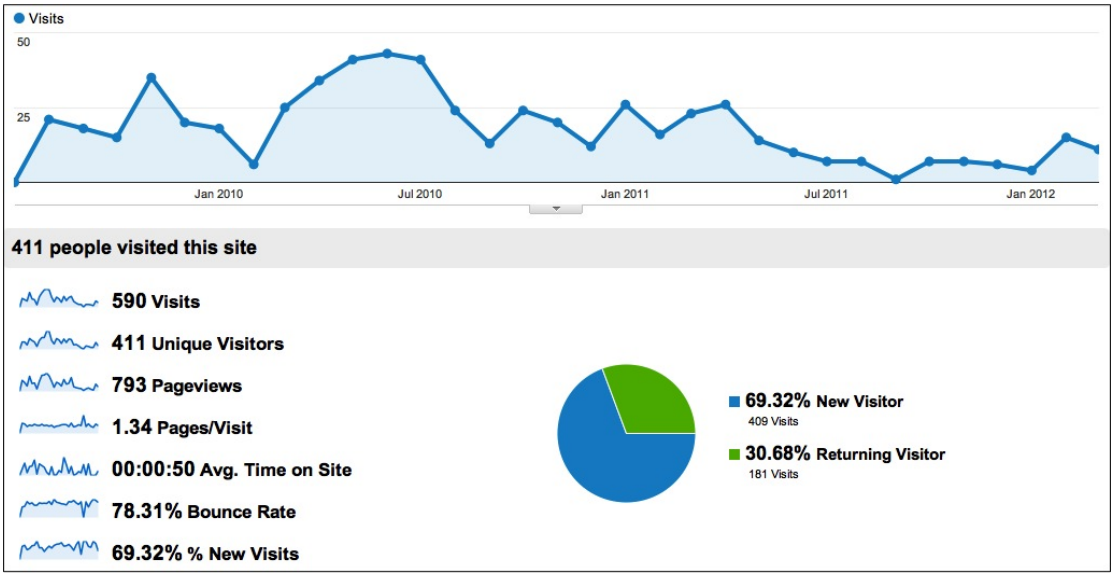


FIGURE 11.1: SWEET Analytics – Overview

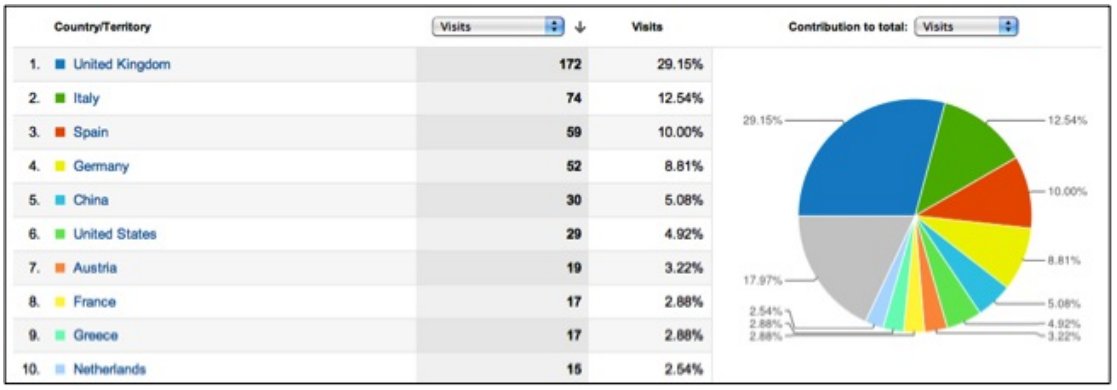


FIGURE 11.2: SWEET Analytics – Country Distribution

Figure 11.2 shows the top ten countries, from which SWEET was accessed. The United Kingdom is number one, which is not surprising, however, it is followed by Italy, Spain and Germany.

These results are more insightful when combined with the list of the top domains used to call the tool (Figure 11.3). These results are very interesting, since they confirm the trend that even if overall there are not that many visitors of SWEET, there are a number of reoccurring ones who used the tool a number of times. This can also be seen as an indication that SWEET managed to gain a certain level of popularity and is continuously used by the same group of people. This data, combined with the fact that one third of the users are returning visitors, points to an overall good uptake of the tool.

The uptake of SWEET is not only reflected by the data collected by Google Analytics but is also evident by the fact that there were continuous requests about the tool via email. For example,

	Service Provider	Visits	Visits
1.	open university	90	15.25%
2.	the open university	30	5.08%
3.	cefriel public subnets	21	3.56%
4.	telecom italia net	13	2.20%
5.	sema group sociedad anonima espana	11	1.86%
6.	multiprotocol service provider to other isp s and end users	10	1.69%
7.	sap ag walldorf	9	1.53%
8.	universidad politecnica de madrid	9	1.53%
9.	comite gestor da internet no brasil	8	1.36%
10.	chungnam national university	7	1.19%

FIGURE 11.3: SWEET Analytics – Frequent Visitors

SWEET was used in Sean Kennedy's[11] PhD thesis on Leveraging the Semantic Web to Automate the Mapping of SOAP Web Services to RESTful HTTP Format and is going to appear as part of a journal article in the Future Internet journal. Furthermore, there were requests from the Database and Information System Group of the University of Brescia, Italy, the Department of Computer Science at the University of Bath, and Atos Origin, Spain to reuse SWEET and include it as part of existing and future work. There were also some requests for developer support and messages from people who looked at the tool. Overall this is very encouraging because it points out that SWEET is of interest for people working in the service area. The few publications and tutorials have already raised some interest in the research community, while the annotation functionalities of the tool seem to adequately address the need for user support in the context of creating semantic Web API descriptions.

11.4.2 Evaluation of Web API Classification Support

Including the type of functionality that the Web API provides as part of the semantic description is key for supporting common tasks such as service discovery and composition, which are predominantly based on finding and using an API with the required support. To this end, in this section we describe the evaluation of the approaches developed for classifying Web APIs. In particular we evaluate the classification solution based on the API HTML documentation in English and a multilingual classification approach that maps API documentation written in different languages to the same category.

[11] Athlone Institute of Technology, Ireland. <SKennedy@AIT.IE>

11.4.2.1 Evaluation of HTML-based Classification

The evaluation of the HTML-based Service Classifier, which uses the API's HTML page as well as a number of related pages[12], was done based on standard precision and recall metrics[13] [GK89]. Table 11.7 shows the results of the classification experiments. As can be seen, by returning a list of five possible classes, with training done only with ProgrammableWeb (cross-validated classifier), there is a 72.5% possibility of returning the correct API category. The results for using the Open Directory Project (OPD classifier) for training show lower accuracy. Despite the fact that usually more training data results in better classification results, this is not the case with the OPD classifier. The main reason behind that is probably the need to do a mapping between the Open Directory classed and the ones done by ProgrammableWeb. OPD has a very rich and hierarchical classification structure that needs to be converted to the rather simple and not so expressive classification taxonomy of ProgrammableWeb. With some further fine-tuning of the classification mappings, the experiment results can probably be improved.

Number of Classes Returned	Cross-validated Classifier (%)	ODP Classifier (%)
1	24.3	9.8
2	39.2	22.0
3	50.4	32.7
4	60.8	44.4
5	72.5	51.5
6	78.5	59.9

TABLE 11.7: Results for Classification Based on the k-Nearest Neighbour

Similarly, the results of the cross-validated classifier, using only data from ProgrammableWeb, are based on assigning a service to only one category, while actually one API can belong to a number of categories. For example, a service for online shopping can be classified as a shopping service but also as an internet service. As a result some correct classifications are in fact considered as incorrect by the evaluation. Therefore, it needs to be further investigated to what extent the current results can be improved by allowing classification to multiple categories. Furthermore, the classification can also be improved by providing better training data, which is validated and contains no misclassifications. The foundation for work in this direction has already been laid by the second Web API survey, which gathers user input about the functionally of the APIs, allowing for selecting multiple classification types, thus creating a validated training set.

[12] All of the analysed documents are in English.

[13] We did not do any ROC analysis [Faw06] in order to be able to make founded statements how our classifier compares to a random one. This should definitively be considered for future evaluations.

In summary, experiments conducted using test data collected from ProgrammableWeb and training data from the Open Directory Project show that the service classifier component is able to determine the correct category among five different classes with more than 70% accuracy.

11.4.2.2 Evaluation of Cross-Lingual Classification

The cross-lingual classification solution has only some preliminary evaluation and tests, which were mainly done in order to investigate the applicability of the approach and to identify points for potential improvements. In particular, the concept detection system was tested on APIs from the geocoding domain. The first phase, which identifies the most similar services, worked quite well, and was able to determine relevant similar Web APIs. Therefore, the classification task was completed successfully. This evaluation needs to be extended to cover further domains, in order to be able to make statements about the precision of the classification approach in general. Our previous experiments with CL-ESA reported in [KZZ11] suggest that the method is able to detect semantically comparable text across languages with high precision (about 0.7 precision at top_{50}) from a 3.5 million large corpus. Given the fact that the size of ProgrammableWeb is smaller and we are classifying only into 54 classes, better results can be expected.

The evaluation tests were very helpful, since they lead to identifying some potential points of improvement. First of all, as is the case with the first classification solution, ProgrammableWeb assigns each API to only one category, while in fact an API can provide functionalities that map to multiple classes. This influences the precision metrics of the evaluation but also distorts the complete classification process, since it is also used to build the background collection. If the background collection is based on missing classification or misclassified services, the classification results will be influenced by that. Therefore, it is important to take steps in order to improve the quality of the data used to train and evaluate the classifier. In particular, this can be done by manually verifying the classes assigned to the APIs and allowing for multiple classes per API.

The second main challenge that became evident through the initial tests was the fact that the documentation URL listed in the directory often does not point to the actual description website but rather to the homepage of the provider, to a common entry page or to an overview page that has very little information about the service. Since this data is not manually validated, the classification is then sometimes based on text that is falsely assumed to be the actual API documentation. Therefore, in order to accurately evaluate the here presented solution, it is important to ensure that the input for the classification is, in fact, the URL that points to the Web API documentation. A final possible point of improvement would be adjusting the list of stop-words, to cover a wide range of works that are API-specific but are not characteristic for the individual API.

Initial evaluation was also done on determining the central concepts, in order to support the annotation of APIs. In contrast to the results for the classification approach, the acquired data for the concept extraction phase indicated that it must be further refined, especially since the returned central concepts were not always relevant. The main reason for this is again the quality of the background collection. Misclassified APIs in the dataset or classification of APIs based on URL not pointing to the actual documentation influence significantly the determining of the central concepts. These limitations can be overcome by hand-picking the representative APIs per category or by ensuring that URLs pointing to the API documentation are correct. Even if improvements still remain to be done, the initial results show that the approach, especially in the context of the classification task, is quite promising.

11.5 Summary

One of the most important features of MSM, the Web API Grounding Model and WAA is that they provide wide coverage, given the heterogeneity of the Web API documentation, in terms of the included information and given details. This ensures that they are widely applicable and not being restricted to a particular sub-group such as, for example, APIs invocable only directly via the URI. We started with a target coverage of 80% and based on the evaluation show that the actual numbers are much higher. Therefore, we can claim that our models for semantically describing Web APIs can support the description of the majority of the APIs.

In the case of the grounding model and WAA, it is also essential to capture all the information that is required for supporting the automation of the corresponding tasks. Therefore, in this chapter we focused on showing that each of the models conforms to the competency questions and the defined design requirements. Furthermore, we revisited the characteristics that we specified and the models need to fulfil, and use the data gathered by the Web API surveys is order to determine, which common service properties are covered and which not. For the Web API Grounding Model and for the Web API Authentication Model, we demonstrated their practical applicability by creating example annotations for Web APIs with different service properties and determining how well they are supported in terms of describing them with each of the models. Therefore, we were able to practically test the models and make sure that they can be used by the authentication and invocation engines.

Finally, we also evaluate the support that we provide for creating semantic Web API descriptions, in terms of the annotation tool and the functionality classification approaches. We evaluate SWEET by showing its usability as part of project use cases and directly using it in hands-on sessions and tutorials. In addition, we log common user actions, in order to determine, which are the most frequently performed annotation tasks, and provide statistics about the numbers and distribution of SWEET's users. We also give evaluation of the approaches developed for

classifying Web APIs. In particular we evaluate the classification solution based on the API HTML documentation and a multilingual classification approach that maps API documentation written in different languages to the same category.

Chapter 12

Conclusions and Future Work

Currently the world of services on the Web is marked by the increased use and popularity of Web APIs. Web APIs are characterised by relative simplicity [PZL08], in comparison to traditional Web services, and a natural suitability for the Web, relying on the interaction primitives provided by the HTTP protocol, with data payloads transmitted directly as part of the HTTP requests and responses. The result is a simpler approach for developing and exposing application interfaces, moving away from the rather complex WS-*specification stack [MSZ01] and returning to adopting the original design principles of the World Wide Web [BL99]. Therefore, Web APIs offer an easy-to-use alternative for simple programmable access to resources, thus enabling third-parties to combine and reuse heterogeneous data coming from diverse services in data-oriented service compositions called mashups. This trend is especially supported by popular social platforms and applications, such as Facebook, Google, Flickr and Twitter, which enable access through Web APIs to some of the resources they hold.

Despite their popularity, as demonstrated throughout this thesis, Web APIs still face a number of challenges. In particular, Web API development is rather autonomous and not guided by standards or guidelines, which results in a wide variety of documentation forms, structures and level of detail, since providers are free to implement and document APIs in any way that they see fit. Furthermore, the majority of the Web APIs are described directly in text as part of HTML pages, which are not meant for automated machine interpretation (in contrast to XML, for example). As a result, currently, Web API use requires extensive manual effort, and client developers have to search through directories, read and interpret the documentation and implement custom solutions that are rarely reusable. Such an approach is time and effort-consuming and will not scale in the context of the growing number of available Web APIs [MPD10a]. Finally, despite some initial efforts [VKVF08] to enable the unified handling of both "traditional" Web services and Web APIs, still the two types of services are stored in separate directories, commonly use different task automation approaches and are rarely deployed in integrated solutions.

The goal of this thesis is to address these challenges and to contribute towards enabling Open Services on the Web, where Web services, APIs, data and Web content can be seamlessly combined and interlinked, without having to differentiate between the separate data sources or the specific technology implementations. In order to achieve this goal we investigated the following four research questions:

- **RQ1:** ***What are the common Web API characteristics?***
- **RQ2:** ***How to describe Web APIs?***
- **RQ3:** ***How to enable a more automated Web API use?***
- **RQ4:** ***How to support the adoption of the new service model?***

We addressed the first question by conducting two thorough studies of the current state of Web APIs (see Chapter 6). The results provide insights about common features and characteristics, establishing trends and the extent of the heterogeneity of the API landscape. The collected data serves as a basis for gaining a deeper understanding of how APIs are exposed on the Web and lays the foundation for developing approaches and solutions towards supporting the use of Web APIs. In particular, the analysis of the studies serves directly as input for answering the second research question, which is tackled in detail in Chapter 7. We take into consideration the data gathered, existing description approaches and present a set of requirements for designing a model, capable of capturing the majority of the existing APIs. The result is the Minimal Service Model (MSM), which represents an operation-based approach towards describing APIs.

Two extensions of MSM are described in Chapters 8 and 9, which provide answers to the question of how a more automated Web API use can be enabled (RQ3). In particular, we focus on invocation and authentication, by defining a set of requirements that need to be met by a model that supports a more automated completion of the corresponding task. In addition, we give details about the two formalisms, provide examples on how they can be applied and evaluate them based on the coverage that they provide and in terms of their use as part of actual working implementations – OmniVoke and two authentication engines.

The final part of the thesis is devoted to the fourth research question and describes SWEET and the solutions provided towards automating some of the annotation tasks, in order to enable a semi-automated process of creating semantic Web API descriptions. In particular, given the here introduced core service model, developers need to be supported by tools that make the creation of Web API descriptions easier. This need is addressed by SWEET and the introduced ontology search and classification solutions.

The following section describes the contributions that were achieved by providing answers to each of the research questions.

12.1 Summary of the Contributions

This section describes our main contributions to the state of the art, in the context of enabling Open Services on the Web through supporting the use of Web APIs.

Contribution 1: We provide an unprecedented in depth analysis of Web APIs and their characteristics, thus contributing towards a clear picture of the actual real world state of APIs on the Web.

A clear understanding of current Web API practices is fundamental for being able to develop improvement solutions with significant impact and take-up. This is crucial for clearly identifying deficiencies and figuring out how existing limitations can be overcome. Therefore, we describe the analysis of the current state of Web APIs, which captures the types of descriptions, how they are exposed, how rich they are and what details they provide. In particular, we give details on:

1.1 Common Web API characteristic and features. The results of the two Web API surveys directly contribute to understanding existing challenges and are a basis for devising solutions and supporting mechanisms. In particular, we collected details and values of different service properties and features, which were used as input for the developed formal description models. Furthermore, the analysis of the current state of Web APIs lays the foundation for the definition of a common description model and served as an input for deriving the properties of the MSM.

1.2 Conclusions on common practices and technologies. Furthermore, we were able to identify trends and frequently used design and solution approaches. For example, we found out that still the majority of the Web APIs are based on the RPC oriented interfaces, as opposed to resource-oriented ones, and this influenced some of the design decisions that were made while defining the description models.

1.3 A web-based survey system. Since the results of the here presented Web API studies have proven to be very useful, we designed and implemented a system that can be customised by other interested parties to gather Web API related details. This is especially relevant with the growing number of APIs, where the analysis task has to be crowd-sourced, in order to cover a larger percentage of the currently available APIs.

Contribution 2: A formal definition of a Web API. Currently there is no commonly accepted definition of a Web API and frequently it is not clear what the underlying principles or technologies are. In order to be able to have a unified and shared understanding of Web APIs, we provide a simple definition. The definition is based on the analysis of common Web API characteristics and serves as the foundation for developing the here introduced formal description models.

Contribution 3: A core service model, in the form of MSM, which enables capturing common API characteristics and providing a foundation for supporting the automation of typical service

tasks. The developed model is based on the results of the Web API analysis and provides a shared overlay over the heterogeneous Web API landscape, therefore, laying common grounds for the development of Web API-based solutions and approaches. In addition, the description model can be applied on top of existing HTML documentation, via mappings to hRESTS tags, and does not necessarily require the creation of new descriptions from scratch, thus conforming with the current trend of providing documentation as part of webpages. Finally, the Web API description model also enables the reuse and adaptation of the wealth of research done in the context of Web services and Semantic Web Services. The core service model is based on the previously defined model as part of hRESTS and was developed in collaboration with a few co-researchers[1]. Still the research carried out in the context of this thesis strongly contributed to improving and consolidating it.

Contribution 4: Web service model extensions for supporting automated invocation and authentication. The core service model is extended with elements that aim to support the automation of specific service tasks, namely invocation and authentication. Therefore, we provide two additional Web API description models:

4.1 The Web API Grounding Model. Invocation is a crucial task in the context of Web API use. Therefore, we developed the Web API Grounding Model as means to enabling its automation.

4.2 The Web API Authentication Model. Authentication is commonly neglected in current Web API approaches. However, it represents a key part in the process of accessing and retrieving API resources. Therefore, we defined authentication-relevant extensions to MSM as part of the Web API Authentication Model.

Both models are based on the analysis of the current state of Web APIs and can be used in combination with MSM or independently of it. In addition, they can be extended to accommodate further properties that might be necessary for certain use cases.

Contribution 5: Support for creating Web API descriptions, in the form of an annotation tool and task-assisting solutions, such as annotation recommendation mechanisms. In particular, we presented:

5.1 SWEET. This is a web application tool that takes as input the HTML documentation of a Web API and provides functionalities for making annotations that conform to MSM and hRESTS/MicroWSMO. The results are the original HTML, enhanced with syntactic and semantic details, and an RDF semantic Web API description.

[1]Carlos Pedrinaci, Dave Lambert, Dong Liu, Jacek Kopecky, Tomas Vitvar, Karthik Gomadam

5.2 Two Web API Classification Approaches. The Web API annotation process can be made easier by automatically completing some tasks that need to be performed frequently. In particular, we focused on determining the type of functionality that the API provides and developed two classification approaches that return a list of possible functionality classes.

In the following section we focus on summarising the conclusions that we reached, based on the conducted work and the achieved contributions.

12.2 Conclusions

In this section we detail our conclusions, which are organised around the four main topics introduced by the research questions.

The general conclusion of our work is that indeed semantic technologies can be used as a basis for contributing to a more integrated Web, where services, data and Web content can be seamlessly combined and interlinked. It particular, they enable the resolving of many of the challenges faced by Web APIs, including the heterogeneity of the API types, implementations and documentation, the lack of machine-interpretable descriptions, and the high degree of manual processing required. A key contribution, therefore, is the development of a formal model for describing Web APIs, which enables the unified handling of the diverse APIs and Web services, alike. Still, there is quite some work to be done before the vision of Open Services on the Web can be practically achieved.

In the remaining of the section we detail our conclusions regarding the current state of Web APIs, the core service model, the specific extensions for providing invocation and authentication support, as well as the tools and approaches developed for enabling the creation of semantic Web API descriptions.

12.2.1 The Current State of Web APIs

The results of the two Web API surveys were very useful in terms of gaining an overview of current characteristics, common practices, and trends. Simply by browsing through the documentation of popular Web APIs, it becomes evident that the diversity in the used documentation forms and structure, as well as the level of provided detail, vary greatly from API to API. The collected data confirmed this impression and provided concrete insights indicating the significance of some of the characteristics and the related issues, at the same time also highlighting important features, which were not recognised initially (such as authentication). Therefore, before any substantial progress and improvement can be made towards supporting and automating the use of Web APIs, we need to reach a deeper understanding of how APIs are developed and

exposed, what kind of documentations are available, how they are represented and how rich these are.

The conducted surveys contribute directly to this goal, by investigating six groups of main features including – general information, type of Web API, input details, output details, invocation details and complementary documentation. We show that currently **Web APIs documentation is solely human-oriented** and not meant for supporting automated API processing. The results of the studies, especially the first one, demonstrate that **REST principles and resource-oriented interfaces are not the driving force behind the current Web API proliferation** and that Web API **documentation is characterised by under-specification**, where important information, such as the datatype and the HTTP methods, is commonly missing.

Simplicity and the trend towards opening data seem to be driving this proliferation, which results in the world of services on the Web being increasingly dominated by Web applications and APIs. These initial observations are confirmed by the second study that also shows that the **Web API environment is very dynamic** with existing APIs being taken offline and new ones being offered. Moreover, by reflecting on the results of the initial survey, we were able to refine the analysed features towards gathering further details and deriving requirements for designing a description model capable of supporting more automated Web API use.

In summary, the impact of the API surveys is twofold:

1. They determine the details that need to be captured by the description model, such as the different types of input parameters or in which part of the HTTP request they are transmitted.
2. Second, they guide decisions on whether certain features should be included or not and in what form, in order to ensure a greater coverage of the model (such as for example, taking a resource or operation-based approach towards describing Web APIs).

12.2.2 The Core Service Model

Regarding the core service model, we can conclude that **providing a unified view on Web APIs is a crucial step** towards overcoming the current heterogeneity of the description forms, characteristics and level of detail, and paving the way towards a more integrated use of services and data on the Web. We use the results of the Web API studies and take into consideration previous work on describing Web APIs, in order to provide a definition of what a Web API is, as used in the context of this thesis. The goal here is to provide a common framework for analysing, describing and using Web APIs.

To this end, we introduce MSM, containing four main classes – *Services*, *Operations*, *MessageContent* and *MessageParts*. The **model is intentionally simple** but still enables capturing the

main service elements, which can be used to add further semantic annotations, by linking service properties to semantic entities. MSM can be used in conjunction with WSMO-Lite, for giving specific semantics to the annotations, or in combination with the Web API Grounding Model and Web API Authentication Model, for providing invocation and authentication support. The core service model, realised through MSM, represents **a major milestone towards enabling the unified handling of Web APIs** and lays the foundation to providing a higher level of task automation.

12.2.3 Towards Automated Web API Invocation and Authentication

Our approach towards enabling more automated Web API use is based on the development of two models – the Web API Grounding Model and the Web API Authentication Model.

The Web API Grounding Model is realised by carefully analysing and gathering the details that are relevant for supporting automated invocation. This is especially important in the context of determining the support provided by the implementation solution and the coverage that it has. As a result, **the model captures invocation-relevant characteristics** and can help to **counteract underspecification**, since it can be used as a reference point by providers in order to determine, which details need to be included as part of the documentation.

The provided invocation support is enhanced with means for capturing authentication details. As demonstrated by the results of the surveys, more than 80% of the APIs require authentication, which makes it a vital part of the invocation process. Therefore, we propose the annotation of authentication information by using the Web API Authentication Model, which overcomes the heterogeneity of authentication approaches and credentials, and provides **the basis for its automated handling as part of the invocation process**.

Both models are practically applicable and have been exploited as part of the OmniVoke invocation and authentication engine. Overall, the Web API Grounding Model and the Web API Authentication Model, in combination with MSM, contribute directly towards establishing guidelines and best practices for creating Web APIs and providing complete documentation. They represent **a comprehensive solution for describing Web APIs and enabling more automated Web APIs use**.

12.2.4 Supporting the Creation of Semantic Web API Descriptions

SWEET represents a major milestone towards supporting the adoption of the description models. In particular, the benefits that come with a lightweight semantic service model cannot be fully used if the creation of the corresponding descriptions is too time- and effort-consuming.

One of the main features of SWEET is that it **hides formalism complexity from the user** and guides him/her through the process of marking the individual service properties and subsequently enhancing these with semantic metadata. As a result, the annotator does not need to be concerned with the model or language specifics and can instead focus on identifying individual properties and linking them to semantic entities. Furthermore, **the annotated HTML can directly be published on the Web**, including the hRESTS/MicroWSMO tags, or can be used to extract a RDF-based description. Without SWEET the user would have to revert to using a text editor and modify the HTML directly, which is a complex, time-consuming and error-prone process.

Still, as highlighted by the user evaluation activities that we carried out, creating semantic Web API descriptions remains a challenging task, even with tool support in the form of SWEET. Therefore, we also introduce a number of solutions aiming towards supporting a semi-automated process of creating Web API descriptions. In particular, this includes integrated semantic entity search via Watson and via customisable search API, as well as two classification approaches based on the type of functionality that the service provides. These solutions contribute towards easing frequently performed API annotation tasks, reducing the manual effort and at the same time lowering the adoption barrier.

12.3 Future Work

In this thesis we have presented three formal models for the semantic description of Web APIs, accompanied by a supporting tool and approaches for automating common annotation tasks, with the ultimate goal of enabling Web API use. However, even if we covered a wide range of issues on this topic, we realise that important research still has to be done in the context of:

- Continued and extended analysis of the current state of APIs on the Web;
- Extension of the invocation work targeted towards supporting the automation of compositions and processes;
- Contributing towards adopting a shared authentication approach;
- Extending SWEET with further annotation support.

This section describes possible future work centred around these four main areas.

12.3.1 Continued Analysis of the State of APIs on the Web

As clearly demonstrated by the results of the two Web API surveys, it is very important to be able to comprehensively grasp the current state of APIs on the Web. Without details about common practices and shared characteristics, any solution would have only limited applicability and its level of practical application can only be estimated or guessed. In this respect we suggest the following lines for future research.

Conducting the survey on a larger scale
In order to continue to gather data on the current state of Web APIs, it would be necessary to conduct the study on a greater scale. There are a number of different options. First, it is possible to release a number of topic-specific surveys, consisting only of a limited number of questions, to a larger group of people. In order to ensure a greater accuracy of the results, the same API should be assessed by a number of participants in order to identify "correct" answers through agreement. In this context, the Web API survey application enables the crowd-sourcing of the surveys, since it is web-based and highly configurable, in terms of the list of questions. Second, a few topic-specific surveys, such as one gathering details on authentication information or the type of API, can be aggregated in order to gain a broader view on the state of the API landscape.

The continuous gathering of further results is crucial not only because this will help us to validate our findings but also to gather insights about new trends and developments. It is important to recognise current solution approaches followed by providers, in order to be able to take them into consideration in the context of improving the here presented solutions.

Analyse the characteristics of the most popular Web APIs
Furthermore, it is important to gain insights by collecting data and analysing the characteristics of the currently most popular Web APIs. These are the APIs, which according to ProgrammableWeb, are used most frequently to build mashups and applications. Therefore, they can be considered as trendsetters in terms of the used description form, features and technologies. Furthermore, we can try to identify the common characteristics and the way, in which they influence the usability of the particular API.

Use survey results for validation
Until now the results of the two Web API surveys have mainly been used as input for designing the description models. However, they also represent data that can be used for evaluation purposes. For example, the categorisation of the APIs can be used to evaluate the accuracy of the classification approaches. This is also true for the manually assigned tags, which can be used to determine the correctness of the automatically identified domain of the description.

12.3.2 Supporting the Invocation of Compositions and Processes

A natural step in further developing our invocation approach is to investigate what details and automation mechanisms are required in order to enable the invocation of compositions and not only of individual APIs. This is an important challenge, since the added value of using multiple sources of data becomes evident precisely through the combination, sorting and filtering of the results of a number of Web APIs. By providing the appropriate extensions to the Web API Grounding Model, OmniVoke can directly be used as an invocation engine for mashups and Web API compositions. This also holds for investigating the possibilities of providing support for process invocation.

12.3.3 Supporting the Adoption of a Shared Authentication Approach

Authentication is currently one topic that is commonly neglected in existing Web APIs approaches and research. Therefore, it is important that its significance is recognised as part of future developments in the area.

Regular updates to the authentication model
Future work in the area of supporting authentication, as a vital part of automating the use of APIs, should focus on updating and revising the Web API Authentication Model, including new and relevant authentication approaches. These changes can be reflected in the implementation solution – the OmniVoke authentication engine. This work is also important in the context of supporting compositions and mashups, where users have to be prompted for the required credentials and if necessary, be redirected to the provider's or trusted party's website.

Encouraging the wider adoption of OAuth
In addition, future work in the area of authentication can be targeted towards supporting the wider use and acceptance of a shared authentication approach, such as OAuth. In particular, the individual semantic Web API descriptions, enriched with annotations about authentication details, can be wrapped and automatically processed to implement only one protocol. For example, a general solution can be based on OAuth authentication against OmniVoke, which stores the individually required credentials and manages them on behalf of the certified user, directly communicating them to the API server upon invocation.

12.3.4 Extending SWEET

The possibilities for extending, improving and adapting SWEET are multifold, including improvements and adjustments of the user interface, options for accessing existing text processing APIs in order to identify and highlight keywords and phrases in the HTML documentation, and

more extended annotation support. Here we describe two main lines of work that might lead to valuable results.

Including further support for automating annotation tasks

SWEET can be extended by providing support for a number of the annotation tasks that can be automated and simply presenting the results to the user for validation. For example, the HTML can be processed in order to directly recognise and highlight the service properties. Furthermore, the corresponding domain can be determined and used to derive a set of ontologies that would be suitable to make annotations. In addition, SWEET can easily be enhanced to include recommendations of individual annotations, or be specifically adopted for a particular domain.

Adapting SWEET to different use cases

The fact that SWEET has already been reused in a number of projects clearly demonstrates that one possible future development is its adaptation to specific use cases. Since the basic functionality of the tool is to annotate HTML and produce RDF, the possibilities for adjustment and adaptation are multifold. It can be used to tag and annotate Website, HTML-based documents or people profiles.

Part V

Appendices

Appendix A

Web API Models

In the appendix we provide more details, which were not included in the main part of the thesis. In particular, we give example descriptions, the complete MSM, Web API Grounding and WAA models in RDF, hRESTS mappings and how-to instructions for SWEET.

A.1 Details on the Minimal Service Model

In this section we give more details on the Minimal Service Model. We include the model in RDF, mappings of the model to hRESTS elements, as well as examples of some semantic descriptions.

Listing A.1 gives the Minimal Service Model in RDF.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix msm: <http://purl.org/msm#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix xml: <http://www.w3.org/XML/1998/namespace>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>.
@prefix dc: <http://purl.org/dc/terms/>.
msm:hasInput a rdf:Property;
        rdfs:domain     msm:Operation;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Input"@en;
        rdfs:range      msm:MessageContent.
msm:hasInputFault a rdf:Property;
        rdfs:domain     msm:Operation;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Input  Fault"@en;
        rdfs:range      msm:MessageContent.
msm:hasMandatoryPart a rdf:Property;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Mandatory Part"@en;
        rdfs:subPropertyOf     msm:hasPart.
msm:hasName a rdf:Property;
        rdfs:domain     msm:MessagePart;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Name"@en;
        rdfs:range      rdf:Literal.
msm:hasOperation a rdf:Property;
        rdfs:domain     msm:Service;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Operation"@en;
        rdfs:range      msm:Operation.
msm:hasOptionalPart a rdf:Property;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Optional Part"@en;
        rdfs:subPropertyOf     msm:hasPart.
msm:hasOutput a rdf:Property;
        rdfs:domain     msm:Operation;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Output"@en;
        rdfs:range      msm:MessageContent.
msm:hasOutputFault a rdf:Property;
        rdfs:domain     msm:Operation;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Output Fault"@en;
        rdfs:range      msm:MessageContent.
msm:hasPart a rdf:Property;
        rdfs:domain     msm:MessagePart;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "has Part"@en;
        rdfs:range      msm:MessagePart.
msm:MessageContent a rdfs:Class;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "Message Content"@en;
        rdfs:subClassOf     msm:MessagePart.
msm:MessagePart a rdfs:Class;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "Message Part"@en.
msm:Operation a rdfs:Class;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "Operation"@en.
msm:Service a rdfs:Class;
        rdfs:isDefinedBy      msm:;
        rdfs:label      "Service"@en.
<http://purl.org/msm> dc:created "2010−03−10"^^xsd:date;
        dc:creator      <http://identifiers.kmi.open.ac.uk/people/carlos−pedrinaci/>,
                        <http://identifiers.kmi.open.ac.uk/people/maria−maleshkova/>,
                        <http://kmi.open.ac.uk/>;

```

```
69      dc:description      "This is a simple ontology that provides the core vocabulary
70      for      capturing service properties"@en;
71      dc:modified     "2011-12-03"^^xsd:date;
72      a     owl:Ontology;
73      rdfs:comment     "This is a description of the Minimal Service Model";
74      rdfs:label       "Minimal Service Model"@en;
75      owl:imports      <http://www.w3.org/ns/sawsdl>,
76                <http://www.wsmo.org/ns/wsmo-lite>;
77      owl:versionInfo      "1.1".
```

LISTING A.1: Minimal Service Model in RDF

Figure A.1 visualises the Minimal Service Model.

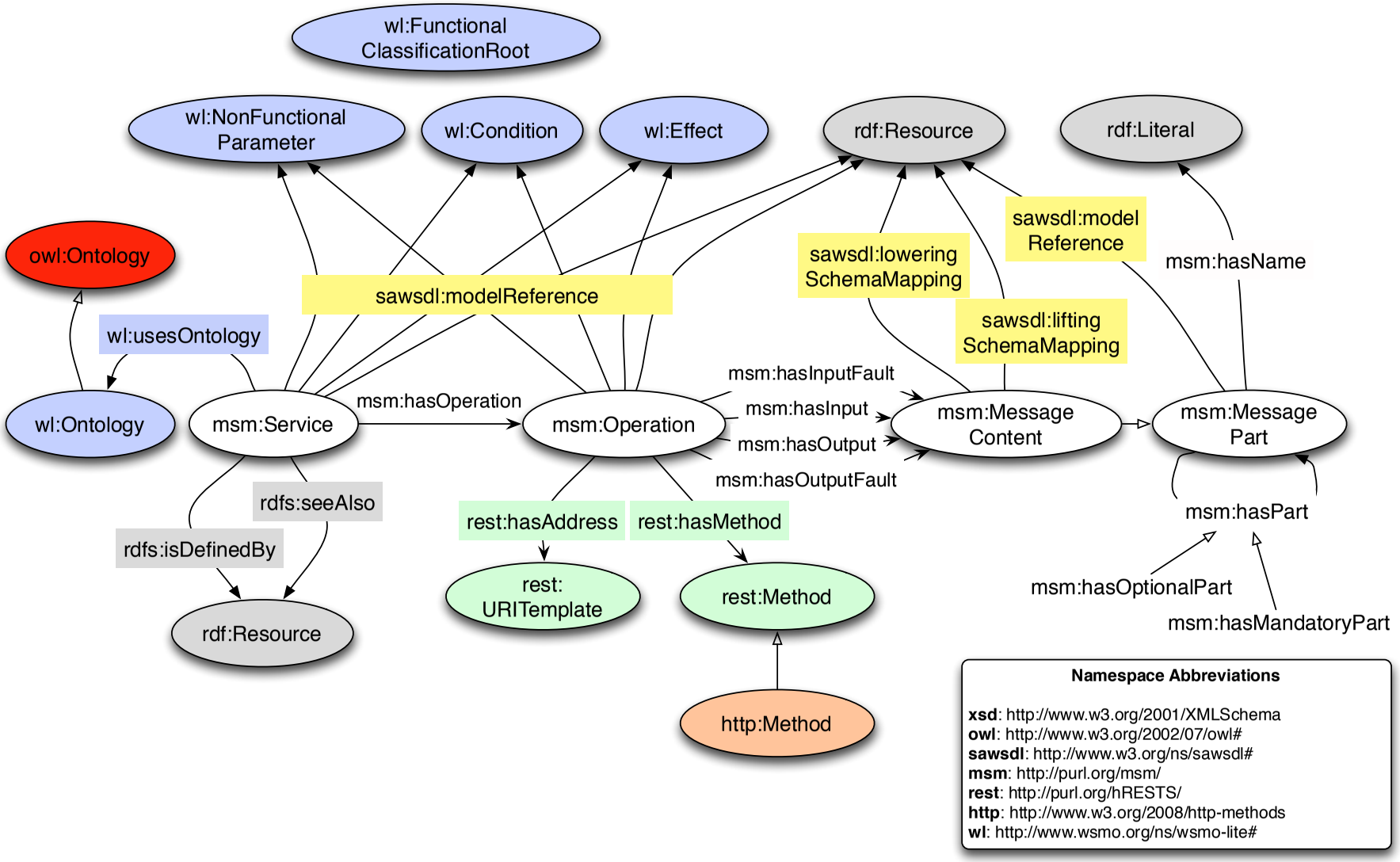


FIGURE A.1: Minimal Service Model

Table A.1 describes how MSM elements are mapped to hRESTS/MicroWSMO tags, thus enabling the creation of MSM-based service representations on the syntactic level. Given the HTML documentation of an API, the hRESTS tags can be used to syntactically structure the documentation, by creating the *service*, *operation*, etc., elements.

MSM Property/Concept	**hRESTS Element**
msm:Service	*class* attribute + *'service'* as a value
msm:hasOperation + msm:Operation	*class* attribute + *'operation'* as a value
msm:hasInput + msm:MessageContent	*class* attribute + *'input'* as a value
msm:hasOutput + msm:MessageContent	*class* attribute + *'output'* as a value
msm:hasPart + msm:MessagePart	*class* attribute + *'parameter'* as a value
msm:hasOptionalPart + msm:MessagePart	*class* attribute + *'parameter'* as a value
msm:hasMandatoryPart + msm:MessagePart	*class* attribute + *'parameter-mandatory'* as a value
msm:hasInputFault + msm:MessageContent	not mapped to hRESTS
msm:hasOutputFault + msm:MessageContent	not mapped to hRESTS
rest:hasAddress + rest:URITemplate	*class* attribute + *'address'* as a value
rest:hasMethod + rest:Method	*class* attribute + *'method'* as a value
msm:hasName + rdf:Literal	*class* attribute + *'label'* as a value
rdfs:isDefinedBy + rdf:Resource	not mapped to hRESTS
rdfs:seeAlso + rdf:Resource	not mapped to hRESTS
sawsdl:modelReference + rdf:Resource	*rel* attribute + *'model'* as a value + *href* to linked URI
not mapped to MSM	*class* attribute + *'default-value'* as a value

TABLE A.1: Mapping MSM to hRESTS Elements

In needs to be pointed out that not all MSM elements need to be mapped to hRESTS and vice versa. For instance the *default-value* is captured only as part of annotations within the HTML. The *default-value* can be nested within a parameter element in order to denote its default value. For example, in the *ArtistGetInfo* operation, the parameter used to specify the language in optional and in many cases the default language is English (even though, not explicitly stated in the example API documentation). In this case the *parameter* would be *lang* and the *default-value* would be English (or, as frequently abbreviated, 'en').

Listing A.2 gives the documentation of a telecommunications Web API that can be used to send SMS messages[1]. The available resources are *messages* and *outbox*, while the set of properties that a message can have are – *guid*, *recipients*, *body*, *title*, *sender*. Some of the message properties are required, while others, such as the *title* are optional.

```
1  HTTP Method: POST
2  Resource: /messages/<guid>/outbox
3  Response: (201) Message Location
4
5  Description: Select the outbox folder to send a SMS Message. The Message will be placed in the sent folder.
6
7  Message Properties:
8  Name        Methods     Value       isRequired   Description
9  guid        GET, POST   string      required     Global unique ID user
10 recipients  GET, POST   [<guid >,..] required     Array of DeviceIds
11 body        GET, POST   string      optional     Message content
12 title       GET, POST   string      optional     Title of message
13 sender      GET, POST   <devid>     required     DeviceIds
```

LISTING A.2: Example of a Resource-based Web API - Send SMS

[1]All the used examples are based on the telecommunications use case from the SOA4All EU project, `http://cordis.europa.eu/project/rcn/85536_en.html`

Listing A.3 shows a simplified example request, where a new message is sent by creating a new message resource in the outbox. The sent message resource properties are in JSON and the used HTTP method is POST. Therefore, based on the described approach for deriving the operation, it will be defined by combining POST and *message* to result in 'postMessage'. The label does not really capture the semantics of the activity, however, the MSM-based description can still be enhanced with, for instance, a classification annotation that states that this is an API for sending SMS.

```
1 HTTP POST /rest/1.0/messages/9764e31d65bf4b2aa3f18b0cdad6e8d6/outbox
2 JSON in the HTTP body:
3 {" recipients ":[" tel :3035551212"], "body":"This is a sms body", " title ":" sms title "," sender":"3035551111"}
```

LISTING A.3: Resource-based Web API - Example Request

Listing A.4 shows a simplified example response, with the confirmation that the resource has been created (Line 1) and a pointer to where it has been created (Line 3).

```
1 Response: 201 CREATED
2 Location:
3 /myhost.com/rest/1.0/messages/3464c32f54vf2d5fd9b71b0cngh1n3d2/sms:10652
```

LISTING A.4: Resource-based Web API - Example Response

Based on this example, we can use the provided details in order to create the following semantic Web API description. Listing A.5 describes a service with a *postMessage* operation.

```
1  @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#> .
3  @prefix msm: <http://purl.org/msm#> .
4  @prefix rest: <http://purl.org/hRESTS#> .
5
6  :service1 rdf:type msm:Service ;
7      rdfs:isDefinedBy "http://developer.ribbit.com/restviews?tid=162"    ;
8      rest:hasAddress "http://rest.ribbit.com:/rest/1.0"       ;
9      msm:hasOperation :postMessage .
10
11 :postMessage rdf:type msm:Operation ;
12     rdfs:label   "sendSMS" ;
13     rest:hasAddress "/messages/{guid}/outbox" ;
14     rest:hasMethod "POST" ;
15     msm:hasInput input ;
16     msm:hasOutput output .
17
18 :input rdf:type msm:MessageContent ;
19     msm:hasMandatoryPart guid ;
20     msm:hasMandatoryPart recipient ;
21     msm:hasPart message .
22
23 :output rdf:type msm:MessageContent ;
24     msm:hasPart messageId .
25
26 :guid rdf:type msm:MessagePart ;
27     sawsdl:modelReference <http://purl.org/atom/ns#id> .
28 :recipient rdf:type msm:MessagePart ;
29     sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_recipient> .
30 :message rdf:type msm:MessagePart ;
31     msm:hasOptionalPart :body ;
32     msm:hasOptionalPart :title ;
33     msm:hasMandatoryPart :sender .
```

```
34
35  :body rdf:type  msm:MessagePart ;
36      sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_Body> .
37  :title   rdf:type  msm:MessagePart ;
38      sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_Title> .
39  :sender rdf:type  msm:MessagePart ;
40      sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_Sender> .
41
42  :messageId rdf:type msm:MessagePart ;
43      sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_Id> .
```

LISTING A.5: Example Semantic Web API Description - Send SMS

The operation has a method *POST*, an *input* and an *output* (Lines 14-16). The *input* consist of three parts, two of which are mandatory (Line 18-21). The third message part contains further parts, two of which are optional and the final one, the sender, is mandatory (Lines 30-33). As exemplified, it is actually very easy to create the description.

We also provide an example based on using the HTTP GET method. Listing A.6 shows a short documentation of the *messages* resource and how it can be used with HTTP GET. This listing also demonstrates that in the previous example with HTTP POST, we could have used the *folder* as an input parameter instead of using a directly fixed path in the address (i.e./messages/{guid}/{folder} vs. /messages/{guid}/outbox). In general, it is up to the annotator to make this decision, and since the initial description already used a fixed path in the address we decided to use it as well.

```
1   HTTP Method: GET
2
3   Resource: /messages/<guid>/<folder>/<msgID>
4   Response: Message Resource
5
6   Description: Ribbit Get details for a specified Message
7
8   Message Properties:
9   Name          Methods     Value        isRequired    Description
10  guid          GET, POST   string       required      Global unique ID user
11  recipients    GET, POST   [<guid >,..] required      Array of DeviceIds
12  body          GET, POST   string       optional      Message content
13   title        GET, POST   string       optional       Title of message
14  sender        GET, POST   <devid>      required      DeviceIds
```

LISTING A.6: Example of a Resource-based Web API - Get Message Details

Listing A.7 gives the semantic Web API description to retrieving the *message* resource with the HTTP GET method, based on the brief documentation in the listing above. The service has an operation *getMessage* with an *input* and an *output* (Lines 11-15). Here the naming of the operation does not capture precisely the semantics of what it actually does. However, as already mentioned, this can be improved by adding a reference to a classification ontology, explicitly stating that this is an operation that gets message details. The *input* consists of three parts, all of which are mandatory (Lines 17-20). These message parts are actually used to specify the values in the address URI pattern (/messages/{guid}/{folder}/{msgID}).

```
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#> .
@prefix msm: <http://purl.org/msm#> .
@prefix rest: <http://purl.org/hRESTS#> .

:service1 rdf:type  msm:Service ;
    rdfs:isDefinedBy "http://developer.ribbit.com/restviews?tid=162"    ;
    rest:hasAddress "http://rest.ribbit.com:/rest/1.0"       ;
    msm:hasOperation :getMessage .

:getMessage rdf:type msm:Operation ;
    rest:hasAddress "/messages/{guid}/{folder}/{msgID}" ;
    rest:hasMethod "GET" ;
    msm:hasInput input ;
    msm:hasOutput output .

:input  rdf:type  msm:MessageContent ;
    msm:hasMandatoryPart  param1 ;
    msm:hasMandatoryPart  param2 ;
    msm:hasMandatoryPart  param3 .

:output rdf:type  msm:MessageContent ;
    msm:hasPart param4 .

:param1 rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://purl.org/atom/ns#id> .
:param2 rdf:type msm:MessagePart ;
    sawsdl:modelReference < http://moguntia.ucd.ie/owl/Datatypes.owl#Folder> .
:param3 rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message_Id> .

:param4 rdf:type msm:MessagePart ;
    sawsdl:modelReference <http://www.w3.org/2004/02/wsa/MessageModel.owl#Message> .
```

LISTING A.7: Example Semantic Web API Description - Get Message Details

We could also define the message parts as optional, however, then the created description would not fit the example documentation, since it would enable the retrieving of further resources (i.e. *folder*) instead of only the message details to a given message Id. Finally, the operation has one output, which is the actual message containing all the message details as well (as opposed to only the message Id, as in the previous example).

A.2 Details on the Web API Grounding Model

In this section we give more details on the Web API Grounding Model. We include the model in RDF and mappings of the model to hRESTS elements.

Listing A.8 gives the Web API Grounding Model in RDF.

```
@prefix dc: <http://purl.org/dc/terms/>.
@prefix http: <http://www.w3.org/2008/http−methods#>.
@prefix msm: <http://purl.org/msm#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix xml: <http://www.w3.org/XML/1998/namespace>.
@prefix rest: <http://purl.org/hRESTS#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>.
rest:acceptsContentType a rdf:Property;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "accepts Content Type"@en;
    rdfs:range    rest:MediaType.
rest:automaticallyExtractedDescription a rdf:Property;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "automatically  Extracted Description"@en;
    rdfs:range    rdf:Literal.
rest:hasAddress a rdf:Property;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "has Address"@en;
    rdfs:range    rest:URITemplate.
rest:hasComment a rdf:Property;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "has Comment"@en;
    rdfs:subPropertyOf    rdf:comment.
rest:hasMethod a rdf:Property;
    rdfs:domain    msm:Operation;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "has Method"@en;
    rdfs:range    rest:Method.
rest:isGroundedIn a rdf:Property;
    rdfs:domain    msm:MessagePart;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "is  Grounded In"@en.
rest:MediaType a rdfs:Datatype;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "Media Type"@en.
rest:OutputFormatParameter a rdfs:Class;
    rdfs:description    "an input  parameter that is  used to determine the format of  the,
    to    use as sawsdl:modelReference on msm:MessagePart)"@en;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "Output Format Parameter"@en.
rest:producesContentType a rdf:Property;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "produces Content Type"@en;
    rdfs:range    rest:MediaType.
rest:URITemplate a rdfs:Class;
    rdfs:isDefinedBy    rest:;
    rdfs:label    "URI Template"@en.
http:Method a rdfs:Class;
    rdfs:subClassOf    rest:Method.
<http://purl.org/hRESTS> dc:created "2010−03−10"^^xsd:date;
    dc:creator    <http://identifiers.kmi.open.ac.uk/people/carlos−pedrinaci/>,
            <http://identifiers.kmi.open.ac.uk/people/maria−maleshkova/>,    <http://kmi.open.ac.uk/>;
    dc:description    "This is  a description  of  the Web API Grounding Model,
    a    simple model for capturing Web API details relevant for  invocation  support"@en;
    dc:modified    "2011−12−03"^^xsd:date;
    a    owl:Ontology;
    rdfs:label    "Web API Grounding Model"@en;
    owl:imports    <http://purl.org/msm>,  <http://www.w3.org/2011/http>, <http://www.w3.org/2011/http−methods>;
    owl:versionInfo    "1.1".
```

LISTING A.8: Web API Grounding Model in RDF

Figure A.2 visualises the Web API Grounding Model.

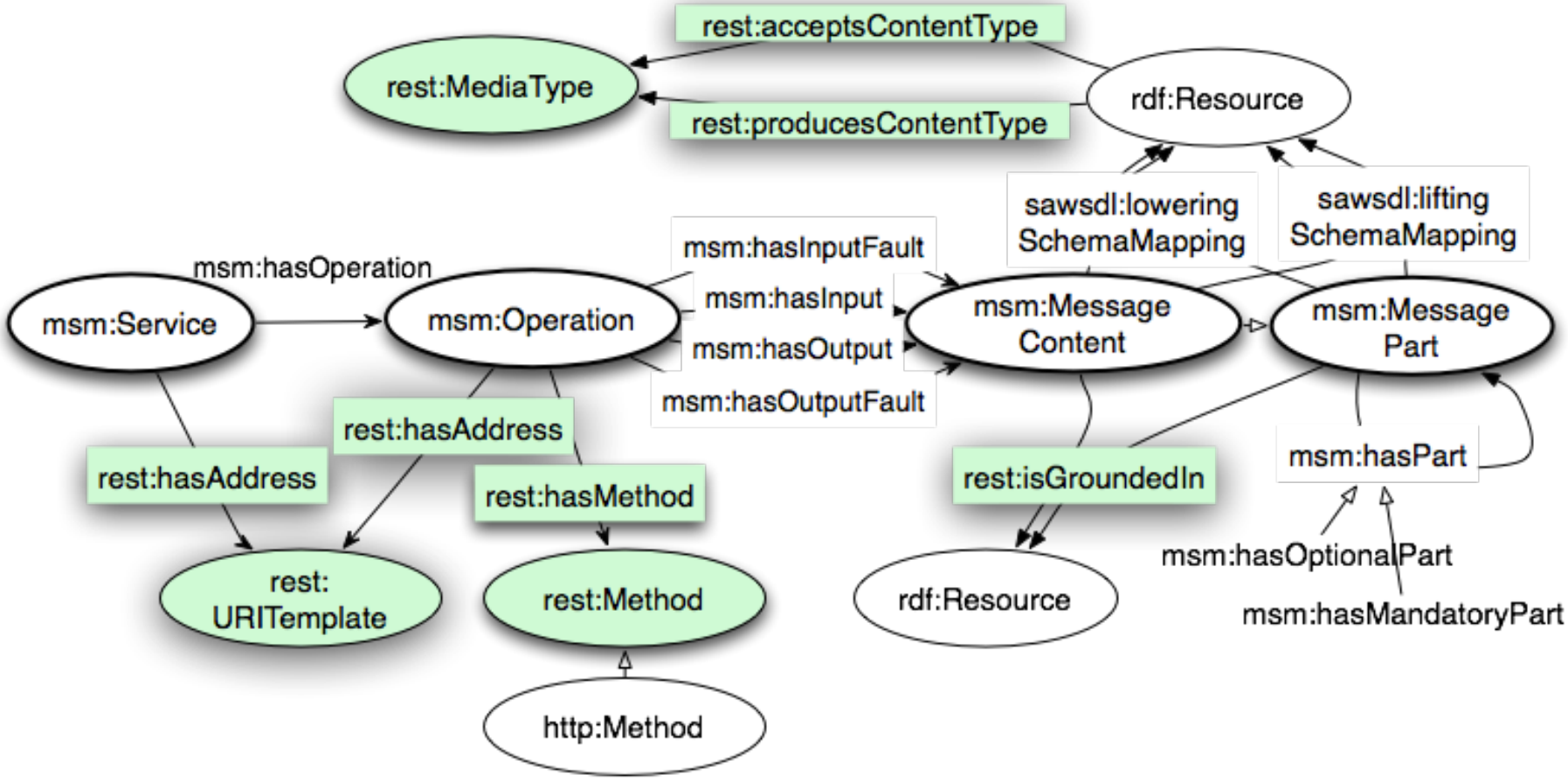


FIGURE A.2: Web API Grounding Model

Listing A.2 contains two additional elements, which were not previously discussed in details – the *OutputFormatParameter* and the *automaticallyExtractedDescription*. These are used to mark content directly within the HTML documentation. As the name suggests, the *OutputFormatParameter* is used to mark parameters that are used to specify the format of the output (XML, JSON, etc.). The *automaticallyExtractedDescription* property is used to mark parts of the documentation, which have been automatically processed and annotated. This element is introduced in order to support mechanisms that process the textual documentation and directly enhance parts of it. Elements marked with this property might, for instance, need validation by the annotator.

Table A.2 describes how the Web API Grounding Model elements are mapped to hRESTS tags.

Grounding Model Property/Concept	**hRESTS Element**
rest:isGroundedIn + rdf:PlainLiteral	*class* attribute + *'grounding'* as a value *title* attribute + name of parameter as value
rest:isGroundedIn + http:body	*rel* attribute + *'grounding'* as a value *href* attribute + *'http://www.w3c.org/2006/http#body'* as a value
rest:isGroundedIn + http:HeaderName	*rel* attribute + *'grounding'* as a value *href* attribute + *'http://www.w3c.org/2006/http#HeaderName'*
rest:hasAddress + rest:URITemplate	*class* attribute + *'address'* as a value
rest:hasMethod + rest:Method	*class* attribute + *'method'* as a value
rest:acceptsContentType + rest:MediaType	*class* attribute + *'content-type'* as a value *title* attribute + media type (e.g application/xml) as value
rest:producesContentType + rest:MediaType	*class* attribute + *'content-type'* as a value *title* attribute + media type (e.g JSON) as value
rest:OutputFormatParameter	*class* attribute + *'parameter-output-format'* as a value
rest:automaticallyExtractedDescription	*class* attribute + *'comment'* as a value

TABLE A.2: Mapping of the Web API Grounding Model to hRESTS Elements

A.3 Details on the Web API Authentication Model

In this section we give more details on the Web API Authentication Model. We include the model in RDF, examples of how annotations can be made with WAA within the HTML and some semantic descriptions of authentication mechanisms, credentials, and the API as a whole.

Listing A.9 gives the Web API Authentication model in RDF.

```
@prefix xml: <http://www.w3.org/XML/1998/namespace>.
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>.
@prefix waa: <http://purl.org/waa#>.
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.
@prefix msm: <http://purl.org/msm#>.
@prefix dc: <http://purl.org/dc/terms/>.
waa:APIKey a rdfs:Class;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "APIKey"@en;
        rdfs:subClassOf     waa:Credentials.
waa:AuthenticationMechanism a rdfs:Class;
        rdfs:description       "The mechanism used to perform authentication"@en;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "AuthenticationMechanism"@en.
waa:composedOf a rdf:Property;
        rdfs:domain     waa:Credentials;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "composedOf"@en;
        rdfs:range     waa:Credentials.
waa:Credentials a rdfs:Class;
        rdfs:description       "The credentials  used to perform the authentication"@en;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "Credentials"@en.
waa:Direct a rdfs:Class;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "Direct"@en;
        rdfs:subClassOf     waa:AuthenticationMechanism.
waa:hasAuthenticationMechanism a rdf:Property;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "hasAuthenticationMechanism"@en;
        rdfs:range     waa:AuthenticationMechanism.
waa:hasInputCredentials a rdf:Property;
        rdfs:domain     waa:AuthenticationMechanism;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "hasInputCredentials"@en;
        rdfs:range     waa:Credentials.
waa:hasName a rdf:Property;
        rdfs:domain     waa:Credentials;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "hasName"@en.
waa:hasValue a rdf:Property;
        rdfs:domain     waa:Credentials;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "hasValue"@en.
waa:HTTPBasic a rdfs:Class;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "HTTPBasic"@en;
        rdfs:subClassOf     waa:AuthenticationMechanism.
waa:HTTPDigest a rdfs:Class;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "HTTPDigest"@en;
        rdfs:subClassOf     waa:AuthenticationMechanism.
waa:isGroundedIn a rdf:Property;
        rdfs:domain     waa:Credentials;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "isGroundedIn"@en.
waa:OAuth a rdfs:Class;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "OAuth"@en;
        rdfs:subClassOf     waa:AuthenticationMechanism.
waa:OAuthConsumerKey a rdfs:Class;
        rdfs:isDefinedBy     waa:;
        rdfs:label       "OAuthConsumerKey"@en;
        rdfs:subClassOf     waa:Credentials.

```

```
69  waa:OAuthConsumerSecret a rdfs:Class;
70      rdfs:isDefinedBy    waa:;
71      rdfs:label      "OAuthConsumerSecret"@en;
72      rdfs:subClassOf    waa:Credentials.
73  waa:OAuthToken a rdfs:Class;
74      rdfs:isDefinedBy    waa:;
75      rdfs:label      "OAuthToken"@en;
76      rdfs:subClassOf    waa:Credentials.
77  waa:OAuthTokenSecret a rdfs:Class;
78      rdfs:isDefinedBy    waa:;
79      rdfs:label      "OAuthTokenSecret"@en;
80      rdfs:subClassOf    waa:Credentials.
81  waa:Password a rdfs:Class;
82      rdfs:isDefinedBy    waa:;
83      rdfs:label      "Password"@en;
84      rdfs:subClassOf    waa:Credentials.
85  waa:realm a rdf:Property;
86      rdfs:domain    waa:AuthenticationMechanism;
87      rdfs:isDefinedBy    waa:;
88      rdfs:label      "realm"@en.
89  waa:requiresAuthentication a rdf:Property;
90      rdfs:domain    msm:Service;
91      rdfs:isDefinedBy    waa:;
92      rdfs:label      "requiresAuthentication"@en;
93      rdfs:range    waa:ServiceAuthentication.
94  waa:ServiceAuthentication a rdfs:Class;
95      rdfs:comment   "Includes all,   some, none as option. Assigned or automatically derived";
96      rdfs:isDefinedBy    waa:;
97      rdfs:label      "ServiceAuthentication"@en.
98  waa:SessionBased a rdfs:Class;
99      rdfs:isDefinedBy    waa:;
100     rdfs:label      "SessionBased"@en;
101     rdfs:subClassOf    waa:AuthenticationMechanism.
102 waa:Username a rdfs:Class;
103     rdfs:isDefinedBy    waa:;
104     rdfs:label      "Username"@en;
105     rdfs:subClassOf    waa:Credentials.
106 waa:wayOfSendingInformation a rdf:Property;
107     rdfs:domain    waa:AuthenticationMechanism;
108     rdfs:isDefinedBy    waa:;
109     rdfs:label      "wayOfSendingInformation"@en.
110 waa:WebAPIOperation a rdfs:Class;
111     rdfs:label      "WebAPIOperation"@en;
112     rdfs:subClassOf    waa:AuthenticationMechanism.
113 <http://purl.org/waa#All>  a waa:ServiceAuthentication.
114 <http://purl.org/waa#Some> a waa:ServiceAuthentication.
115 <http://purl.org/waa#None> a waa:ServiceAuthentication.
116 <http://purl.org/waa>  dc:created "2010−05−10"^^xsd:date;
117     dc:creator     <http://identifiers.kmi.open.ac.uk/people/carlos−pedrinaci/>,
118             <http://identifiers.kmi.open.ac.uk/people/maria−maleshkova/>,
119             <http://kmi.open.ac.uk/>;
120     dc:description     "This is  a description  of  the Web API Authentication model,
121     a    simple model for capturing Web API−related authentication information"@en;
122     dc:modified    "2011−12−03"^^xsd:date;
123     a    owl:Ontology;
124     rdfs:label      "Web API Authentication Model"@en;
125     owl:imports    <http://purl.org/msm>;
126     owl:versionInfo     "1.1".
```

LISTING A.9: Web API Authentication Model in RDF

Figure A.3 visualises the Web API Authentication Model.

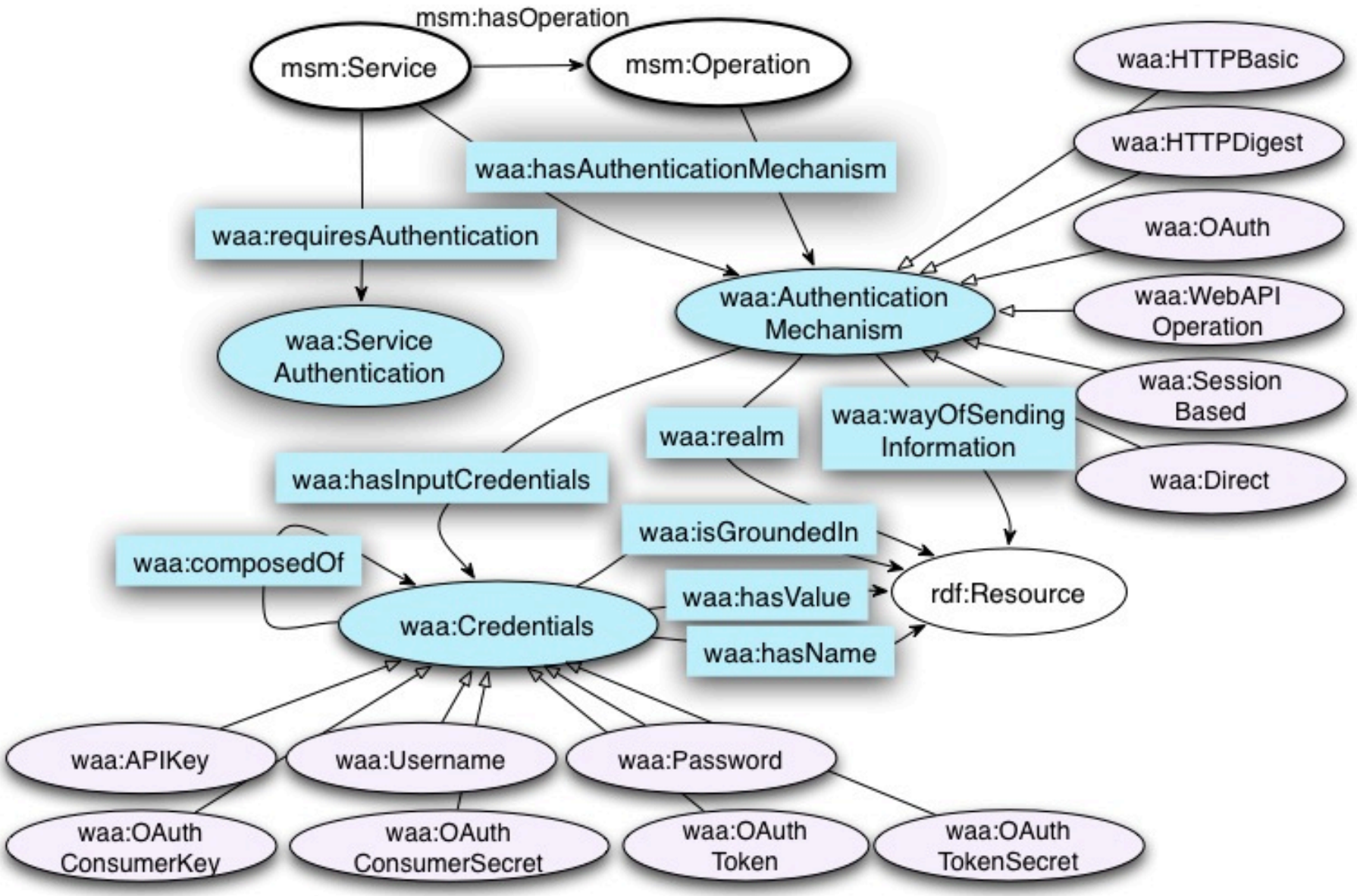


FIGURE A.3: Web API Authentication Model

Table A.3 describes how WAA-based annotations can be made within the HTML documentation. We use the model reference tag, as provided by SAWSDL. All of the mappings are based on using model references and there are no newly introduced hRESTS tags.

Authentication Model Property/Concept	**hRESTS Element**
waa:requiresAuthentication + waa:ServiceAuthentication	*rel* attribute + *'model'* as a value *href* attribute + http://purl.oclc.org/waa#All OR + http://purl.oclc.org/waa#Some OR + http://purl.oclc.org/waa#None
waa:hasAuthenticationMechanism + waa:AuthenticationMechanism	*rel* attribute + *'model'* as a value *href* attribute + URI to a AuthenticationMechanism instance

TABLE A.3: Mapping of the Web API Authentication Model to hRESTS Elements

Listing A.10 shows an example HTML documentation, with inserted authentication details.

```
1 <div class="service" id="service1"><h1>Last.fm Web Services</h1>
2 <a rel="model" href="http://purl.oclc.org/waa#All">
3 <div class="operation" id="op1"><h2><span class="label">artist.getInfo</span></h2>
4 <a rel="model" href="http://purl.oclc.org/NET/WebApiAuthentication/LastFm">
5 <span class="address">http://ws.audioscrobbler.com/2.0/?method=artist.getinfo...</span>
6 <div class="input" id="input1">...</div>
7 <div class="output" id="output1">Artist</div></div></div>
```

LISTING A.10: Web API Authentication Information through hRESTS

Listing A.11 shows how this instance of the *AuthenticationMechanism* class looks like. As can be seen, the capturing of authentication information with the provided Web API authentication ontology is very simple and easy to apply.

```
1 @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
2 @prefix waa: <http://purl.oclc.org/waa#> .
3 <http://purl.oclc.org/NET/WebApiAuthentication/LastFm> rdf:type waa:Direct ;
4   waa:realm <http://www.last.fm/api/> ;
5   waa:hasInputCredentials <http://purl.oclc.org/NET/WebApiAuthentication/LastFmAPIKey> ;
6   waa:wayOfSendingInformation waa:ViaURI .
```

LISTING A.11: Example Instance of the AuthenticationMechanism Class

Finally, listing A.12 shows how example authentication *waa:APIKey* credentials can look like.

```
1 @prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
2 @prefix waa: <http://purl.oclc.org/waa#> .
3 <http://purl.oclc.org/NET/WebApiAuthentication/LastFmAPIKey> rdf:type waa:APIKey ;
4   waa:hasName "Last.fm API Key" ;
5   waa:hasValue "myAPIKey29813719823918273" .
```

LISTING A.12: Example Credentials for APIKey

Listing A.13 shows a complete semantic Web API description with authentication information.

```
@prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#> .
@prefix msm: <http://purl.org/msm#> .
@prefix rest: <http://purl.org/hRESTS#> .
@prefix waa: <http://purl.org/waa#> .

:lastfmService a msm:Service ;
  rdfs:isDefinedBy <http://www.last.fm/api/show?service=267> ;
  rest:hasAddress "method=artist.getinfo&artist={p1}  &api_key={p2}"^^rest:URITemplate ;
  waa:requiresAuthentication waa:All ;
  msm:hasOperation :ArtistGetInfo .
:ArtistGetInfo  a msm:Operation ;
    msm:hasInput :ArtistGetInfoInput ;
    rest:hasAddress "http://ws.audioscrobbler.com/2.0/?"^^rest:URITemplate .
    waa:hasAuthenticationMechanism :lastfmAuth .
:lastfmAuth a waa:Direct ;
    waa:realm <http://www.last.fm/api/> ;
    waa:hasInputCredentials :api_key ;
    waa:wayOfSendingInformation waa:ViaURI .
:api_key a waa:APIKey ;
    waa:isGroundedIn "p2" .
:ArtistGetInfoInput  a msm:MessageContent ;
    msm:hasPart :artist .
:artist  a msm:MessagePart ;
    rest:isGroundedIn "p1"^^rdf:PlainLiteral   .
```

LISTING A.13: Example Authentication Details Annotation I

Listing A.14 shows some further examples.

```
@prefix : <http://iserve.kmi.open.ac.uk/resource/services/e8f9548e−bbed−43fe−9d8a−71b7fdef9da#> .
@prefix rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf−schema#> .
@prefix msm: <http://purl.org/msm#> .
@prefix rest: <http://purl.org/hRESTS#> .
@prefix waa: <http://purl.org/waa#> .

:lastfmService a msm:Service ;
  rdfs:isDefinedBy <http://www.last.fm/api/show?service=267> ;
  rest:hasAddress "method=artist.getinfo&artist={p1}&api_key={p2}"^^rest:URITemplate;
  waa:requiresAuthentication waa:All ;
  msm:hasOperation :ArtistGetInfo .
:ArtistGetInfo  a msm:Operation;
  msm:hasInput :ArtistGetInfoInput;
  rest:hasAddress "http://ws.audioscrobbler.com/2.0/?"^^rest:URITemplate.
waa:hasAuthenticationMechanism :lastfmAuth.

#credentials based on API key
:lastfmAuth a waa:Direct;
  waa:realm <http://www.last.fm/api/> ;
  waa:hasInputCredentials :api_key ;
  waa:wayOfSendingInformation waa:ViaURI .
:api_key a waa:APIKey ;
  waa:isGroundedIn "p2" .

#alternative for the credentials based on username and password
:lastfmAuth a waa:Direct;
  waa:realm <http://www.last.fm/api/> ;
  waa:hasInputCredentials :a, :b ;
  waa:wayOfSendingInformation waa:ViaHTTPHeader.
:a a waa:Username;
  waa:isGroundedIn "h1" .
:b a waa:Password;
  waa:isGroundedIn "h2" .

```

```
36  #OAuth example
37  :yelpAuth a waa:OAuth ;
38    waa:realm <http://www.ye.lp/> ;
39    waa:hasInputCredentials :consumer_key, :consumer_secret, :token, :token_secret .
40  :consumer_key a waa:OAuthConsumerKey .
41  :consumer_secret a waa:OAuthConsumerSecret .
42  :token a waa:OAuthToken .
43  :token_secret a waa:OAuthTokenSecret .
```

LISTING A.14: Example Authentication Details Annotation II

Appendix B

Supporting Tools

In this appendix we provide more details and how-to instructions for SWEET.

B.1 Using SWEET to Make Annotations

In this section we provide an example, as well as a sample hands-on session for SWEET.

Listing B.1 shows an example service description annotated with MicroWSMO by using SWEET. Line 2 uses the `model` relation to indicate that the service searches for events, while line 10 associates the input parameter *username* with the class *Username*. The lowering schema for the recipient is also provided in line 11.

```
1  <div class="service" id="s1"><h1>happenr API</h1>
2  <a rel="model" href="http://example.com/events/getEvents">
3  <span class="label">Happenr </span>has two main methods to call "getEvents" and ...</a>
4  <p>All operations should be directed at http :// happenr.3scale.net/</p>
5  <h2>Example usage</h2>
6  <span class="address">http://happenr.3scale.ws/webservices/getEvents.php?user_key=xxx</span>
7  <p>where the userkey is the key issues with the signup you made.</p>
8  <div class="operation" id="op1"><h2><span class="label">getEvents </span>Method</h2>
9  <span class="input">
10 <h3><a rel="model" href="http://example.com/data/onto.owl#Username">username</a>
11 (<a rel="lowering" href="http://example.com/data/event.xsparql">lowering</a>)</h3>
12 <p>Your username that you received from Happenr in order to query this webservice.</p>
13 <h3><a rel="model" href="http://example.com/data/onto.owl#Password">password<a>
14 (<a rel="lowering" href="http://example.com/data/event.xsparql">lowering</a>)</h3>
15 <p>Your password that you received from Happenr in order to query this webservice.</p>
```

LISTING B.1: Example Web API Description - Happenr

B.1.1 Hands-on with SWEET

In this section we give a hands-on session, based on three tasks.
Hands-On Sessions Outline:

- Task 1 - Service Discovery
- Task 2 - Web API Semantic Annotation
- Task 3 - Semantics-based Web Service Search

Task 1: Service Discovery (Finding Web APIs and Web Services)
Imagine the following three scenarios and try to find suitable Web APIs and Web services on the Web that could help you.
Scenario 1: Mapping/Geocoding API
The client is looking for a service that provides mapping (showing map, routes, directions) or geocoding (resolving an address to a geographical location) functionalities.
Scenario 2: US Geocoding Request
The client is looking for a service to geocode US addresses (e.g. lookup the geographic location of a postal addresses). The license status of services should be ignored for the search.
This service has the following inputs:

- US postal address of type address: A structured US postal address (street, house number, city, state, five letter zip code). We assume that the client is able to provide the address in unstructured format, too.

This service has the following outputs:

- Geographic location of type geographic point: Latitude and longitude of the given address (or of the bounding box corresponding to the given address).

Scenario 3: US City Data Request
The client is looking for services that provide information about a given US location using city and state as input. The client is most interested in the zip code(s), area code(s) and the geographic location. Should a city have multiple zip codes and area codes, all are requested. If the three data items mentioned above are provided, the service is considered ideal, but even if none of these are provided, the service should be considered relevant as long as some other information about the city (map of the city, population, time zone, current events, traffic situation... the more the better) is offered.
This service has the following inputs:

- US city name of type city: A valid US city name
- US state code of type province: The US state that the city is located in

This service has the following outputs:

- Zip codes, list of type postal code: The 5-digit zip code(s) for the city
- Area code(s), list of type area code: The 3-digit area code(s) for the city
- Geographic location of type geographic area: The geographic location (lat/lng, either point or bounding box) of the city

Task 2: Web API Semantic Annotation

With the help of SWEET try to annotate some of the Web APIs below:

- Geo Services (www.earthtools.org)
- Geocoder (nearby.org.uk)
- Geocoding API : Reverse Geocoding API : Cross Street Intersection Geocoding API (geocoder.ca)
- Directions Web Service API (www.geosmart.co.nz)
- Point of Interest (POI) Web Service - Version 2 API Documentation (www.geosmart.co.nz)
- Geonames Web Services (www.geonames.org)
- Google Static Maps (code.google.com)

Step-by-step Guide for Annotating the GeoNames API Documentation

Open SWEET – http://sweetdemo.kmi.open.ac.uk/soa4all/MicroWSMOeditor.html

Open GeoNames WebService documentation – http://www.geonames.org/export/web-services.html

1. Annotate the CountryCode operation
2. Create Service property
3. Rename it to "GeoNames"
4. Create Operation property
5. Rename it to "CountryCode"
6. Create Input
7. Create Output
8. Create Address
9. Rename address property (not the label itself) to http://api.geonames.org/countryCode

10. Create HTTP Method
11. Rename method property (not the label itself) to "GET"
12. Create Parameter "lat"
13. Select the "lat" string then double-click on Parameter
14. Rename the parameter to "lat"
15. Create Parameter "long"
16. Create Parameter "username"
17. Create Parameter "soCode"
18. On Service – add model reference to:
    – http://www.service-finder.eu/ontologies/ServiceCategories#Maps%20and%20Geography
    – http://www.service-finder.eu/ontologies/ServiceOntology#Free
19. On Input – add model reference to:
    – lat – http://www.w3.org/2003/01/geo/wgs84_pos#lat
    – long – http://www.w3.org/2003/01/geo/wgs84_pos#long
    – username – http://purl.oclc.org/NET/WebApiAuthentication#Username
20. On Output – add model reference to:
    – isoCode – http://www.geonames.org/ontology#countryCode
21. Input lowering – http://people.kmi.open.ac.uk/ning/Schema/GeoNames/CountryCodeLowering.txt
22. Output lifting – http://people.kmi.open.ac.uk/ning/Schema/GeoNames/CountryCodeLifting.txt
23. Publish the Description:
    – Look at the annotated HTML by saving it to your local machine
    – Look at the generated RDF by exporting it to you local machine
24. Publish the description in the semantic Web service repository iServe http://iserve-dev.kmi.open.ac.uk/iserve
    – credential to use: maria, maria
    – Note the service ID

Task 3: Discovering Web APIs

Now that you have uploaded the annotated version of the Web APIs, try to use iServe to answer the queries from Task 1. You are encouraged to use the SPARQL querying window in iServe Browser to this end.

Look at you service description:

– http://iserve-dev.kmi.open.ac.uk/iserve/page/services/YOURSERVICEID

– http://iserve-dev.kmi.open.ac.uk/iserve/page/services/db4b646a-4665-4337-9626-4669cc8bce56
Find it by using the taxonomy tree on the left.

Search directly with a SPARQL query, based on functionality (Listing B.2).

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX wsl:<http://www.wsmo.org/ns/wsmo-lite#>
PREFIX sawsdl:<http://www.w3.org/ns/sawsdl#>
PREFIX msm:<http://cms-wg.sti2.org/ns/minimal-service-model#>

SELECT ?s WHERE {
?s rdf:type  msm:Service.
?s sawsdl:modelReference ?modelref .
?modelref rdfs:subClassOf <http://www.service-finder.eu/ontologies/ServiceCategories#Maps\%20and\%20Geography> .
```

LISTING B.2: Sample SPARLQ Query - Search Based on Functionality

Search for free services (Listing B.3).

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX wsl:<http://www.wsmo.org/ns/wsmo-lite#>
PREFIX sawsdl:<http://www.w3.org/ns/sawsdl#>
PREFIX msm:<http://cms-wg.sti2.org/ns/minimal-service-model#>

SELECT ?s WHERE {
?s rdf:type  msm:Service.
?s sawsdl:modelReference <http://www.service-finder.eu/ontologies/ServiceOntology#Free> . }
```

LISTING B.3: Sample SPARLQ Query - Search for Free Services

Look for a service with particular input (Listing B.4).

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX wsl:<http://www.wsmo.org/ns/wsmo-lite#>
PREFIX sawsdl:<http://www.w3.org/ns/sawsdl#>
PREFIX msm:<http://cms-wg.sti2.org/ns/minimal-service-model#>

SELECT ?s WHERE {
?s rdf:type  msm:Service.
?s msm:hasOperation ?op .
?op msm:hasInput ?in .
?in sawsdl:modelReference <http://www.w3.org/2003/01/geo/wgs84_pos#long> .
?op msm:hasInput ?in1 .
?in1 sawsdl:modelReference <http://www.w3.org/2003/01/geo/wgs84_pos#lat> .
```

LISTING B.4: Sample SPARLQ Query - Search Based on Input

Look for a service with particular input/output (Listing B.5).

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf−schema#>
PREFIX xsd:<http://www.w3.org/2001/XMLSchema#>
PREFIX rdf:<http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX wsl:<http://www.wsmo.org/ns/wsmo−lite#>
PREFIX sawsdl:<http://www.w3.org/ns/sawsdl#>
PREFIX msm:<http://cms−wg.sti2.org/ns/minimal−service−model#>

SELECT ?s WHERE {
?s rdf:type  msm:Service.
?s msm:hasOperation ?op .
?op msm:hasInput ?in .
?in sawsdl:modelReference <http://www.w3.org/2003/01/geo/wgs84_pos#long> .
?op msm:hasInput ?in1 .
?in1 sawsdl:modelReference <http://www.w3.org/2003/01/geo/wgs84_pos#lat> .
?op msm:hasOutput ?out.
?out sawsdl:modelReference <http://www.geonames.org/ontology#countryCode> .
```

LISTING B.5: Sample SPARLQ Query - Search Based on Input and Output

B.2 The Web API Survey System

The experience of the first study, where all the results were collected in a single spreadsheet, raised the need for a supporting tool that would speed up the analysis process and let the survey participant focus on the API documentation and not on how the collected inputs are stored and structured. The presented system is implemented as a Web application that can be easily configured and without redeployment be used to conduct different surveys in parallel. It enables the crowdsourcing of

B.2.1 Survey Model and Setup

The Web API survey system was designed to ease the gathering of Web API details by enabling people from distributed locations to provide input, thus supporting the crowd-sourcing of the Web API analysis task. The precise formulation of the questions, the selection of available answers as well as the provisioning of examples, enable easy completion of the survey, also without much prior knowledge in the area of Web APIs.

The survey system implements all API characteristics used as part of the second survey, however, it can be configured to show only a desired subset in order to collect information about a restricted number of features. For example, the system can be used to gather details related only to classification or only to authentication. The result is a flexible and customisable Web application that can be distributed to different groups of participants in order to serve a variety of research interests.

The data collected by the survey system is stored in a triplestore in RDF [Hay04]. We use a very simple underlying model, where each survey entry in an instance of the *ServiceRepositorySurveyEntry* class with namespace *http://kmi.open.ac.uk/web-api*. A survey entry has a set of properties given in detail in Table B.1. The properties can be divided into seven groups, including properties relating to API general information, input details, output details, invocation details, type of API, additional documentation and survey details.

In particular, the survey properties are used for analysing the collected data. For example, the *scope* determines as part of which survey the set of triples was created. It can be used to distinguish between the data gathered as part of different surveys but that is stored by the same system deployment in the same triplestore. Similarly, the *debug* can be used to see a summary of the user input that has been submitted upon completing the survey. The following section describes in detail how the system was implemented and how the here described survey model was put into practice.

ID	Property Name	ID	Property Name
1	apiId	20	input-hasDatatype
2	api-name	21	input-hasSessionInfo
3	api-lastUpdated	22	input-hasCodedParams
4	api-desc	23	input-links
5	documentationUrl-isCorrect	24	input-hasRequiredParams
6	apiClasses	25	input-hasOptionalParams
7	tags	26	input-isGroundedIn
8	numberOfOperations	27	input-hasAltParams
9	survey-canBeCompleted	28	input-hasBoolParams
10	user	29	input-isObject
11	debug	30	output-format
12	scope	31	output-hasSchema
13	hasDateTime	32	auth-transMedium
14	webapi-type	33	authreq
15	desc-hasInvocUri	34	auth
16	isHttpMethodDefined	35	auth-example
17	uri-hasTemplates	36	err-has
18	uri-hasVersionInfo	37	err-useHttpErrs
19	uri-hasQueryParams	38	example-resp
		39	example-req

TABLE B.1: Web API Survey Model

B.2.2 Survey System Implementation

The Web API survey system is realised in the form of a Web application and has a very traditional architecture, consisting of three main components – user interface, data processing and data storage. The user interface is designed as a sequence of Web forms, which gather user input and present information about the API that is being analysed[1]. There are no dynamic elements used, however, the individual parts of the forms can be shown or not depending on the current configuration. As a result the user interface implements visualisation of all Web API features but the specific combination of features to use can be changed by setting particular parameter values. Figure B.1 shows the first pages of the user interface.

The data processing component transforms the input gathered via the user interface and prepares it for storage. Similarly, it also retrieves and transforms the general Web API information, such as name, description, and URL to the documentation, and passes it to the UI. The actual processing is very minimal and involves mostly data formatting and adjustment. Finally, the data storage component includes two repositories. The first one implements the survey model as part of a triplestore, where all the gathered input is collected. The second one stores data obtained from the ProgrammableWeb website, including details about each API that are presented as part of the survey, either as informative details or as features that need validation. We used a

[1]The presented general information is based on the ProgrammableWeb directory.

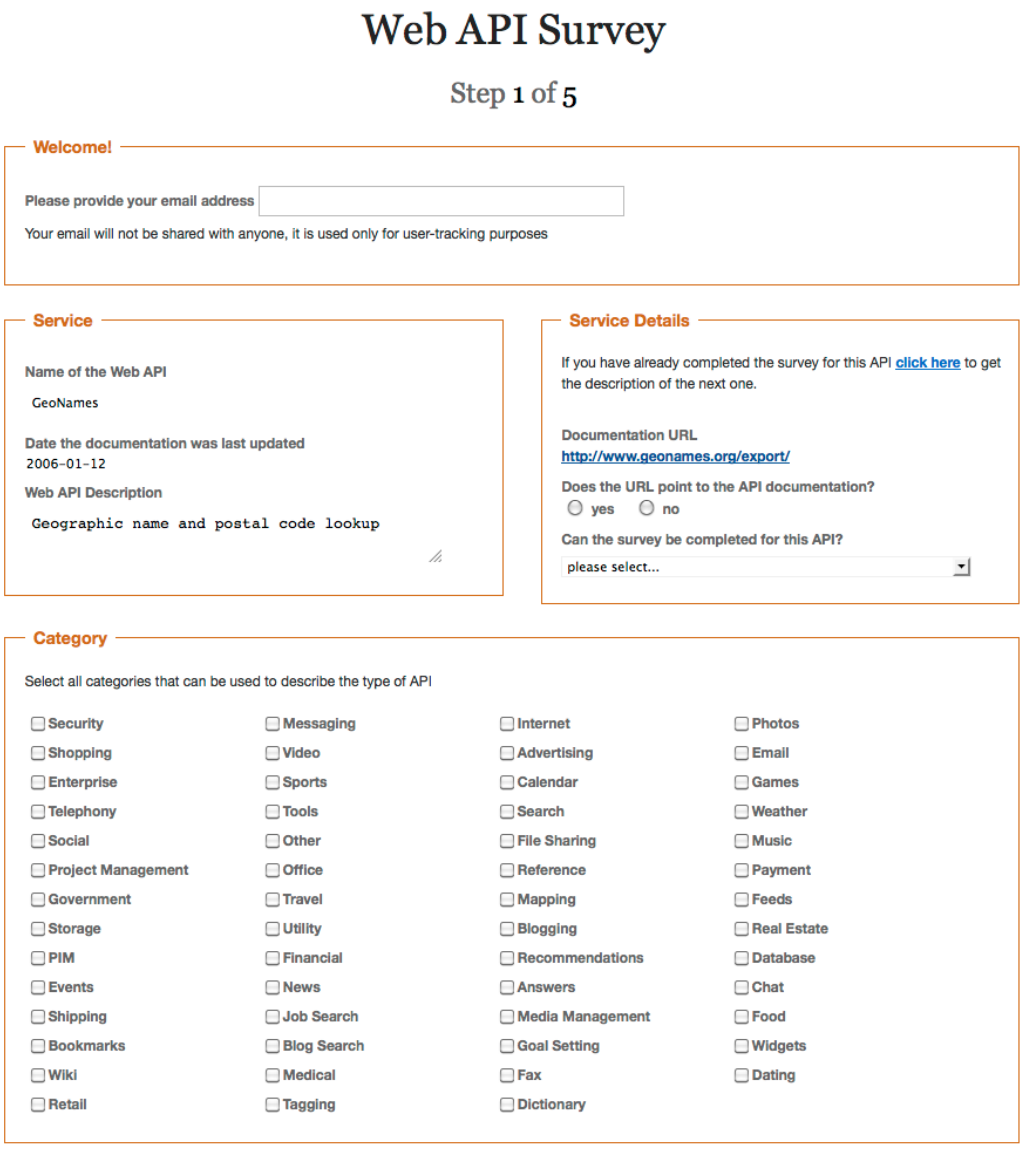


FIGURE B.1: Web API Survey System - Form 1

Sesame server[2] for the data storage. The Web application is based on JavaServer Pages (JSP) and Java, while the user interface is implemented with Cascading Style Sheets (CSS) and HTML. The complete code of the survey system project is available under `https://github.com/mmale/WebAPISurvey`.

As already mentioned, the system can be easily configured to show or hide different parts of the survey. Figure B.2 shows the second page of the Web application. Parts of this page, or the complete page can be hidden by setting the corresponding values to the 'remove' parameter – *rm*. For example, *?rm=%23auth-fieldset* hides the box containing authentication-relevant questions. The *rm* parameter needs to be set to the escape value of the IDs of the fields that need to

[2] `http://www.openrdf.org/`

be hidden (remove #section1,#fieldsetQuery will be ?rm=%23section1%2C%23fieldsetQuery). This is because hide/remove is realised with the help of a CSS jQuery selector.

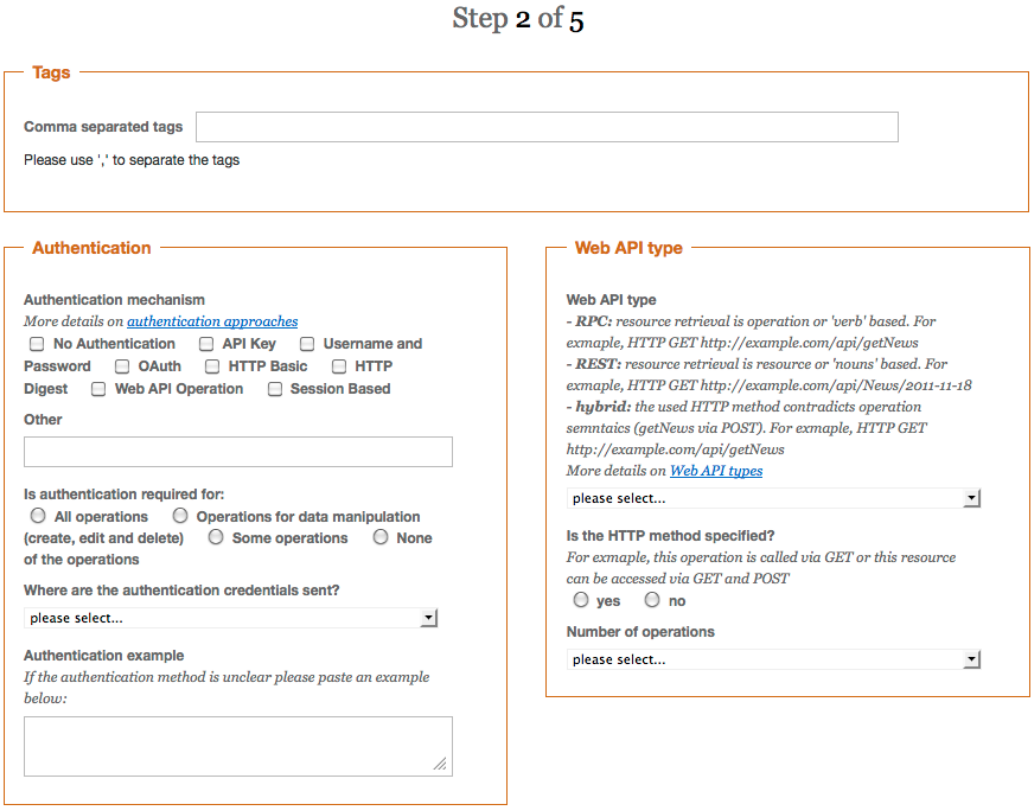


FIGURE B.2: Web API Survey System - Form 2

Figure B.3 and Figure B.4 show the fields for collecting input and output details correspondingly. However, there are three possible ways of determining for which API the survey should be completed. First, the default setting can be used by loading the survey directly via the URL without any parameters. In this case the next API, for which the survey has not been completed yet, will automatically be selected and returned. In addition, it can be configured that each APIs has a number of survey entries, for example, at least two, before moving on to the next API (i.e., at least two people have to complete the survey for one particular API). However, the value of this parameter needs to be set before the survey system is deployed.

Second, the survey can be completed for a particular API, by passing the API ID in the dataset, storing the data collected from ProgrammableWeb. For example, a specific value for the URI would be *?URI=http://localhost:8080/lpw/resource/apis/aol-video*.

Third, the particular API to use in the survey can be determined by directly calling it via the index value (for example, *?index=6*). The index is the sequence number of the API in the repository, when all APIs are sorted in an ascending order. Since the index might change when the collection of APIs is modified, it is best to retrieve a particular API over the URI ID. The

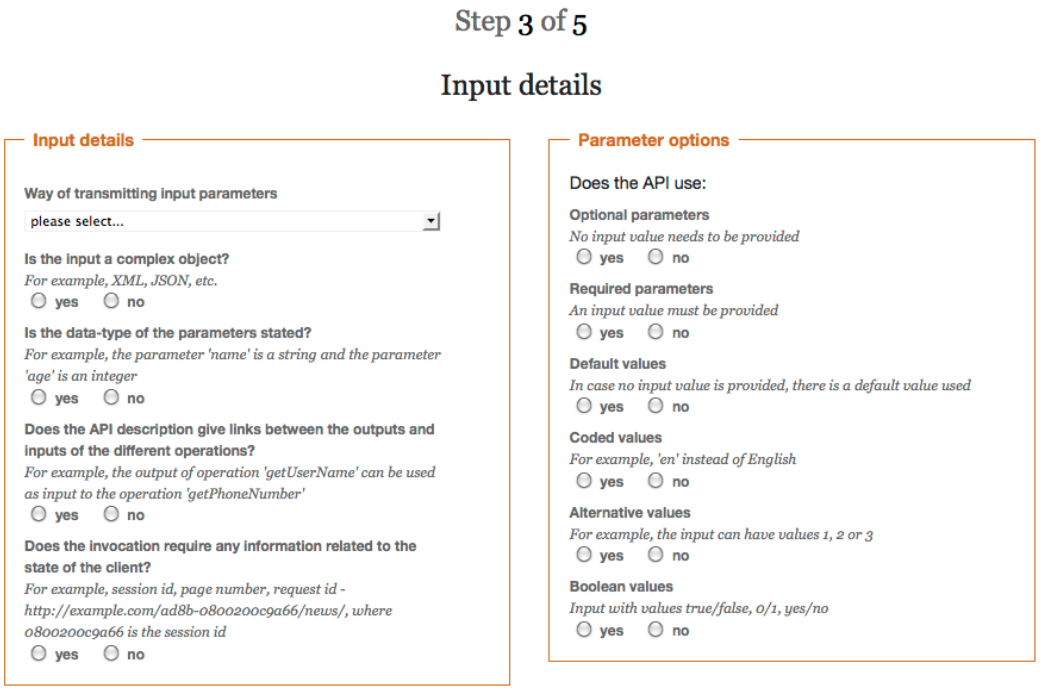


FIGURE B.3: Web API Survey System - Form 3

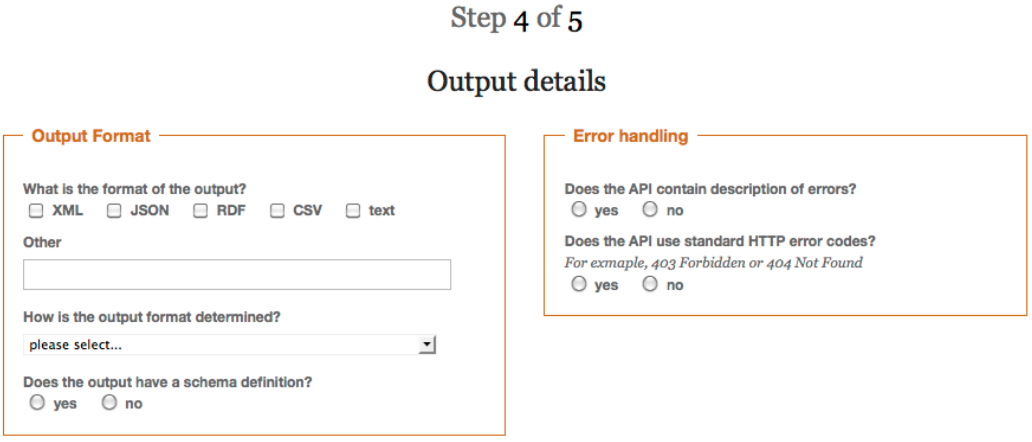


FIGURE B.4: Web API Survey System - Form 4

option for explicitly selecting the APIs to participate in the survey is especially useful for cases when only a specific subset needs to be analysed. For example, we used this option in order to collect details about the APIs used in the first study, by creating a list based on the Excel table from two years ago and completing the survey for one API after the other.

Finally, Figure B.5 shows the last page of the survey, which is used to submit the answers. The user has no way of directly seeing what exactly is posted to the repository, however, in some cases it might be necessary to verify that the stored details are correct. Therefore, we have included the *debug* parameter, which if set to *show*, displays a summary of the user input (for

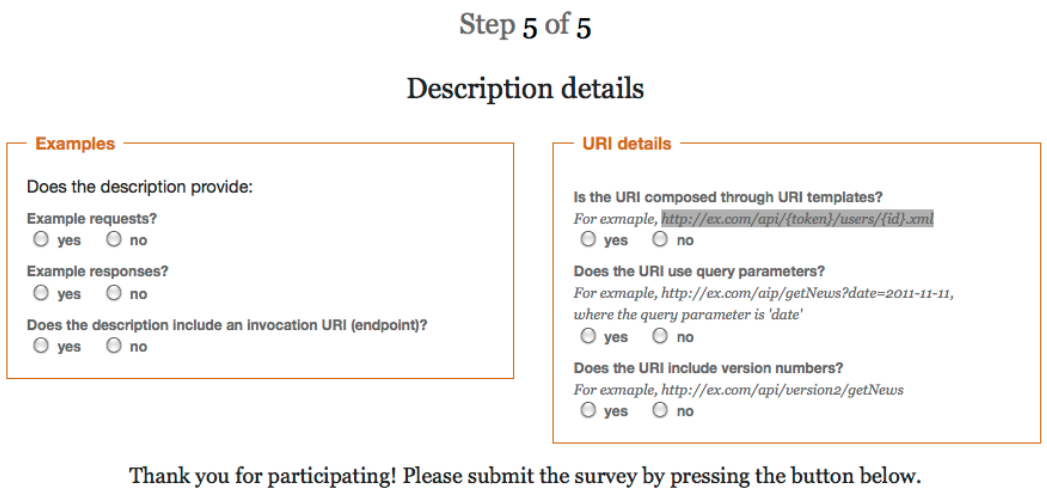


FIGURE B.5: Web API Survey System - Form 5

example, *?URI=http://localhost:8080/lpw/resource/apis/aol-video&debug=show*). This option is useful when testing the system but also when a particular configuration of the system needs to be verified. It is important to point out that in the cases where no value is set for a particular property in the survey model, i.e., the question in the survey is not answered, the corresponding tripe in the repository is not created. Therefore, a *SurveyEntry* instance resulting from completing only the classification section would have less property instances than one created as a result of filling out the complete set of forms and questions. In order to avoid having to deal with a very long parameterised URL, we recommend that it is stored as a PURL or an abbreviated URL, so that it is easier to distribute it to the survey participants.

So far the Web API survey system has been used in two use cases. First, we used it to collect the data during the second Web API survey. In fact, the system was developed especially in order to enable the completion of that study. Second, we configured the system to collect only classification-relevant details, which were used to reflect on the correctness of the classification done by ProgrammableWeb but also to evaluate our Web API classification approach, based on the HTML documentation, described in Chapter 10. Since the survey system is provided in the form of a Web application, it is very easy to distribute the link to the survey to a large group of people and use crowd-sourcing for analysing a large number of APIs. In the future we hope to benefit from the potential of the survey system in a wider range of use cases.

Bibliography

[ABH+01] A. Ankolekar, M. H. Burstein, J. R. Hobbs, O. Lassila, et al. DAML-S: Semantic markup for web services. In *Semantic Web Working Symposium (SWWS)*, 2001.

[ABMP08] Ben Adida, Mark Birbeck, Shane McCarron, and Steve Pemberton. RDFa in XHTML: Syntax and Processing, W3C Recommendation, 2008.

[ACKM04] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications*. Springer-Verlag, Berlin, 2004.

[ADG+09] José Luis Ambite, Sirish Darbha, Aman Goel, Craig A. Knoblock, Kristina Lerman, Rahul Parundekar, and Thomas Russ. Automatically constructing semantic web services from online sources. In *The Semantic Web – ISWC 2009: 8th International Semantic Web Conference, Chantilly, VA, USA*, pages 17–32, 2009.

[AFM+05] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagaraja, Marc-Thomas Schmid, Amit Sheth, and Kunal Verma. Web Service Semantics - WSDL-S. Technical report, World Wide Web Consortium, http://www.w3.org/Submission/WSDL-S/, November 2005.

[AGDR03] R. Akkiraju, R. Goodwin, P. Doshi, and S. Roeder. A method for semantically enhancing the service discovery capabilities of UDDI. *IIWeb*, pages 87–92, 2003.

[AKKP08] Waseem Akhtar, Jacek Kopecky, Thomas Krennwallner, and Axel Polleres. XSPARQL: Traveling between the XML and RDF worlds and avoiding the XSLT pilgrimage. In Manfred Hauswirth, Manolis Koubarakis, and Sean Bechhofer, editors, *Proceedings of the 5th European Semantic Web Conference*, LNCS, Berlin, Heidelberg, June 2008. Springer Verlag.

[All11] Subbu Allamaraju. Describing RESTful applications. *infoq*, December 2011.

[AMG+10] G. Álvaro, I. Martínez, J.M. Gómez, F. Lecue, C. Pedrinaci, M. Villa, and G. Di Matteo. Using SPICES for a better service consumption, 2010.

[AMQ08] E. Al-Masri and M. H. Qusay. Investigating Web Services on the World Wide Web. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pages 795–804, 2008.

[AW10] R. Alarcón and E. Wilde. RESTler: crawling RESTful services. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *Proceedings of the 19th international conference on World Wide Web, WWW '10*, pages 1051–1052. ACM, 2010.

[BAM08] Devis Bianchini, Valeria Antonellis, and Michele Melchiori. Flexible semantic-based service matchmaking and discovery. In *Proceedings international conference on World Wide Web*, volume 11, pages 227–251, 2008.

[BBB+05] Steve Battle, Abraham Bernstein, Harold Boley, Benjamin Grosof, Michael Gruninger, Richard Hull, Michael Kifer, David Martin, Sheila McIlraith, Deborah McGuinness, Jianwen Su, and Said Tabet. Semantic Web Services Language (SWSL). Member submission, W3C, 2005.

[BCG07] Christian Bizer, Richard Cyganiak, and Tobias Gauss. The RDF Book Mashup: From Web APIs to a Web of Data. In *3rd Workshop on Scripting for the Semantic Web*, June 2007.

[BCH08] Chris Bizer, Richard Cyganiak, and Tom Heath. How to Publish Linked Data on the Web, 2008.

[BCPS05] Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, and Rita Scognamiglio. An approach to support web service classification and annotation. In *Proc. of IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 138–143. IEEE Press, 2005.

[BEH+02] Erol Bozsak, Marc Ehrig, Siegfried Handschuh, Andreas Hotho, Alexander Maedche, Boris Motik, Daniel Oberle, Christoph Schmitz, Steffen Staab, Ljiljana Stojanovic, Nenad Stojanovic, Rudi Studer, Gerd Stumme, York Sure, Julien Tane, Raphael Volz, and Valentin Zacharias. KAON - towards a large scale semantic web. In Kurt Bauknecht, A. Min Tjoa, and Gerald Quirchmayr, editors, *Proceedings of the Third International Conference on E-Commerce and Web Technologies EC-Web 2002*, Lecture Notes in Computer Science, pages 304–313. Springer, 2002.

[Bel08] Michael Bell. Introduction to service-oriented modeling. *Service-Oriented Modeling: Service Analysis, Design, and Architecture*, page 3, 2008.

[BHBL09] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.

[Bis12] Stefan Bischof. Optimising XML-RDF data integration - a formal approach to improve XSPARQL efficiency. In Elena Simperl, Philipp Cimiano, Axel Polleres, Oscar Corcho, and Valentina Presutti, editors, *ESWC*, volume 7295 of *Lecture Notes in Computer Science*, pages 838–843. Springer, 2012.

[BL99] T. Berners-Lee. *Weaving the Web*. Harpur, San Francisco, 1999.

[BlCC+06] Tim Berners-lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the 3rd International Semantic Web User Interaction Workshop*, 2006.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic syntax. http://tools.ietf.org/html/rfc3986, 2005.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[Bre09] F. Breitling. A standard transformation from XML to RDF via XSLT. *CoRR*, abs/0906.2291, 2009.

[BvHH+04] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, et al. OWL Web Ontology Language Reference, W3C Recommendation, 2004.

[CD07] Pádraig Cunningham and Sarah Jane Delany. k-Nearest neighbour classifiers, 2007.

[CDM+04] Liliana Cabral, John Domingue, Enrico Motta, Terry R. Payne, and Farshad Hakimpour. Approaches to semantic web services: an overview and comparisons. In Christoph Bussler, John Davies, Dieter Fensel, and Rudi Studer, editors, *ESWS*, volume 3053 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2004.

[Cer02] Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. OŔeilly Media, February 2002.

[CFNO04] M. Champion, C. Ferris, E. Newcomer, and D. Orchard. Web Services Architecture. W3C Working Draft, World Wide Web Consortium, 2004.

[CGK+03] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services,

Version 1.1. Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation, 2003.

[Chi07] E. Chinthaka. REST and web services in WSDL 2.0. http://www.ibm.com/developerworks/webservices/library/ws-rest1/, 2007.

[CHvRR04] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI Spec Technical Committee Draft 3.0.2. Oasis committee draft, OASIS, 2004.

[CLL+10] Y. Chen, J. Li, Y. Lv, H. Qin, and L. Zhang. DRWSC - to simplify dynamic invocation for RESTful web services. In *ICSES, IEEE Press*, 2010.

[Clo08] T. Close. Web-key: Mashing with permission. In *Proceedings of Web 2.0 Security and Privacy*, 2008.

[Cod08] Codehaus. XFire. http://xfire.codehaus.org/, 2008.

[Con11] World Wide Web Consortium. HTTP vocabulary in RDF 1.0, working draft may 2011. http://www.w3.org/TR/HTTP-in-RDF10/M, 2011.

[CS03] J. Cardoso and A. Sheth. Semantic e-Workflow Composition. *Journal of Intelligent Information Systems*, 21(3):191–225, November 2003.

[CSM02] Jorge Cardoso, Amit Sheth, and John Miller. Workflow Quality Of Service. Technical report, LSDIS Lab, Computer Science, Universtity of Georgia, Athens GA, USA, March 2002.

[Dai12] Robert Daigneau. *Service design patterns: fundamental design solutions for SOAP/WSDL and RESTful web services*, volume 37. Addison-Wesley (E), 2012.

[Dav05] Mark Davydov. SOA adventures, Part 1: Ease Web services invocation with dynamic decoupling. http://www.ibm.com/developerworks/webservices/library/ws-soa-adventure1/, April 2005.

[dBLK+05] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel. The Web Service Modeling Language (WSML). http://www.wsmo.org/TR/d16/d16.1/v0.2/, 2005.

[DCG+08] J. Domingue, L. Cabral, S. Galizia, V. Tanasescu, A. Gugliotta, B. Norton, and C. Pedrinaci. IRS-III: A Broker-based Approach to Semantic Web Services. *Journal of Web Semantics*, 2008.

[DM08] John Domingue and David Martin. Semantic Web Services. Tutorial on Semantic Web Services colocated with the ISWC 2008, 2008.

[dM11] M. d'Aquin and E. Motta. Watson, more than a Semantic Web search engine. *Semantic Web Journal*, 2011.

[DMD07] M. Dzbor, E. Motta, and J. Domingue. Magpie: Experiences in supporting Semantic Web browsing. *Web Semantics: Science, Services and Agents on the WWW*, 2007.

[dMD+08] Mathieu d'Aquin, Enrico Motta, Martin Dzbor, Laurian Gridinoc, Tom Heath, and Marta Sabou. Collaborative semantic authoring. *IEEE Intelligent Systems*, 23(3):80–83, 2008.

[DMS+01] M. J. Duftler, N. K. Mukhi, A. Slominski, E. Slominski, and S. Weerawarana. Web services invocation framework (WSIF). In *OOPSLA Workshop on Object Oriented Web Services*, 2001.

[DSK+07] M. Dimitrov, A. Simov, M. Konstantinov, L. Cekov, and Momtchev V. WSMO Studio - a semantic web services modelling environment for WSMO (system description). In *Proceedings of the 4th European Semantic Web Conference (ESWC). Number 4519 in LNCS*, 2007.

[dSM+08] M. d'Aquin, M. Sabou, E. Motta, S. Angeletou, L. Gridinoc, V. Lopez, and F. Zablith. What can be done with the Semantic Web? an overview of Watson-based applications. In *5th Workshop on Semantic Web Applications and Perspectives*, 2008.

[ELM03] Susana Eyheramendy, David D. Lewis, and David Madigan. On the Naive Bayes model for text categorization, 2003.

[ERG02] Peter Werner Eklund, Nataliya Roberts, and Steve P. Green. OntoRama: Browsing an RDF ontology using a hyperbolic-like browser. In *The First International Symposium on CyberWorlds (CW2002)*, Theory and Practices, pages 405–411. IEEE press, 2002.

[Faw06] Tom Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[FB96] N. Freed and N. Borenstein. Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies, 1996.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1 (RFC 2616). Request For Comments, 1999. Available at `http://www.ietf.org/rfc/rfc2616.txt`, accessed 7 July 2006.

[FHBH99] J. Franks, P. Hallam-Baker, and J. Hostetler. HTTP authentication: Basic and digest access authentication rfc2617. *The Internet Society*, 1999.

[Fie00] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[FKZ08] D. Fensel, M. Kerrigan, and M. (eds.) Zaremba. *Implementing Semantic Web Services: the SESA Framework*. Springer, 2008.

[FL07] J. Farrell and H. Lausen. Semantic Annotations for WSDL and XML Schema (SAWSDL). W3C recommendation. http://www.w3.org/TR/2007/REC-sawsdl-20070828/, August 2007.

[FLP+06] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domigue. *Enabling Semantic Web Services. The Web Service Modeling Ontology*. Springer, Berlin, Heidelberg, 2006.

[FN10] Florian F. Fischer and Barry Norton. D3.4.6 microWSMO v2 – defining the second version of microWSMO as a systematic approach for rich tagging. Soa4all project deliverable, 2010.

[GAS11] T. Gottron, M. Anderka, and B. Stein. Insights into explicit semantic analysis. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1961–1964. ACM, 2011.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[GK89] Michael D. Gordon and Manfred Kochen. Recall-precision trade-off: A derivation. *JASIS*, 40(3):145–151, 1989.

[Gom98] A. Gomez-Perez. Knowledge sharing and reuse. In J. Liebowitz, editor, *Handbook of Expert Systems*. CRC, 1998.

[GRN+08] K. Gomadam, A. Ranabahu, M. Nagarajan, A. Sheth, and K. Verma. A faceted classification based approach to search and rank Web APIs. In *Proceedings of the 2008 IEEE International Conference on Web Services*, 2008.

[Gru93] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, pages 5(2):199–220, 1993.

[Gru95] T. R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human and Computer Studies*, 43(5/6):907–982, 1995.

[Gru07] Tom Gruber. Collective Knowledge System: Where the Social Web meets the Semantic Web. *Journal of Web Semantics*, 6 (1):4–13, 2007.

[GSM+09] D. Gessler, G. Schiltz, G. May, S. Avraham, C. Town, D. Grant, and R. Nelson. SSWAP: A simple semantic web architecture and protocol for semantic web services. *BMC Bioinformatics*, 10:309, 2009.

[Had06] M. J. Hadley. Web application description language (WADL). Technical report, Sun Microsystems. https://wadl.dev.java.net, November 2006.

[Hau09] Michael Hausenblas. Exploiting Linked Data to Build Web Applications. *IEEE Internet Computing*, 13(4):68–73, 2009.

[Hay04] Patrick Hayes. RDF Semantics. Recommendation, World Wide Web Consortium, February10 2004.

[HB11] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data SpaceTheory and Technology*, volume Volume 1. Morgan & Claypool Publishers, 2011.

[HDM+04] F. Hakimpour, J. Domingue, E. Motta, L. Cabral, and Y. Lei. Integration of OWL-S into IRS-III. In *First AKT Workshop on Semantic Web Services*, 2004.

[Hen97] J. Hendler. A Little Semantics Goes a Long Way, 1997.

[HJK04] A. Hess, E. Johnston, and N. Kushmerick. ASSAM: A tool for semi-automatically annotating semantic web services. In *Proceedings of the 3rd International Semantic Web Conference (ISWC)*, 2004.

[HK03] Andreas Hess and Nicholas Kushmerick. Automatically attaching semantic metadata to Web Services. In *Proceedings of IIWeb*, pages 111–116, 2003.

[HK04] A. Hess and N. Kushmerick. Machine Learning for Annotating Semantic Web Services. *In AAAI Spring Symposium on Semantic Web Services*, March 2004.

[HLSE08] P. Hasse, H. Lewen, R. Studer, and M. Erdmann. The NeOn ontology engineering toolkit. 2008.

[HS02] Siegfried Handschuh and Steffen Staab. Authoring and annotation of web pages in CREAM. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 462–473, New York, NY, USA, 2002. ACM.

[JMH+07] Robert Jaeschke, Ro Marinho, Andreas Hotho, Lars Schmidt-Thieme, and Gerd Stumme. Tag recommendations in folksonomies. In *PKDD*, pages 506–514. Springer, 2007.

[JSR03] JSR-101 Expert Group. Java API for XMLBased RPC, Version 1.1. http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec10, 2003.

[KAO02] KAON. http://kaon.semanticweb.org/, 2002.

[Kay07] Michael Kay. XSL transformations (xslt) version 2.0. World Wide Web Consortium, Recommendation REC-xslt20-20070123, January 2007.

[KC06] R. Khare and T. Celik. Microformats: a pragmatic path to the semantic web (poster). In *Proceedings of the 15th international conference on World Wide Web*, 2006.

[KG05] B. Parsia A. Kalyanpur and J. Golbeck. SMORE - semantic markup, ontology, and RDF editor, 2005.

[KG07] D. Kohlert and A. Gupta. Java API for XML-Based Web Services, Version 2. http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html, 2007.

[KGV08] Jacek Kopecky, Karthik Gomadam, and Tomas Vitvar. hRESTS: an HTML Microformat for Describing RESTful Web Services. *In Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence (WI-08)*, 2008.

[KHL+07] Akrivi Katifori, Constantin Halatsis, George Lepouras, Costas Vassilakis, and Eugenia Giannopoulou. Ontology visualization methods – a survey. *ACM Comput. Surv.*, 39(4):10, 2007.

[KKRKS07] Ulrich Küster, Birgitta König-Ries, Michael Klein, and Mirco Stern. DIANE: A matchmaking-centered framework for automated service discovery, composition, binding, and invocation on the Web. *Int. J. Electron. Commerce*, 12:41–68, December 2007.

[KKRM05] Michael Klein, Birgitta König-Ries, and Michael Mussig. What is needed for semantic service descriptions: A proposal for suitable language constructs. *Int. J. Web Grid Serv.*, 1:328–364, December 2005.

[KMTF07] Mick Kerrigan, Adrian Mocan, Martin Tanler, and Dieter Fensel. The Web Service Modeling Toolkit - An integrated development environment for semantic web services (system description). In *Proc. of the 4th European Semantic Web Conference (ESWC)*, pages 789–798, 2007.

[KNM10] Reto Krummenacher, Barry Norton, and Adrian Marte. Towards linked open services and processes. In Arne-Jørgen Berre, Asunción Gómez-Pérez, Kurt Tutschku, and Dieter Fensel, editors, *Future Internet - FIS 2010 - Proceedings of the Third Future Internet Symposium, Berlin, Germany, September 20-22,*

2010, volume 6369 of *Lecture Notes in Computer Science*, pages 68–77. Springer, 2010.

[KPS+04] L. Kagal, M. Paolucci, N. Srinivasan, G. Denker, T. Finin, and K. Sycara. Authorization and privacy for semantic web services. *IEEE Intelligent Systems*, page 19:4, July 2004.

[KV07] Jacek Kopecky and Tomas Vitvar. D11v0.2 WSMO-Lite. Technical report, WSMO Working Draft, 2007.

[KV08] J. Kopecky and T. Vitvar. MicroWSMO. CMS Working Draft. http://www.wsmo.org/TR/d38/v0.1/20080219/, February 2008.

[KVF09] Jacek Kopecky, Tomas Vitvar, and Dieter Fensel. D3.4.3 MicroWSMO and hRESTS. Deliverable for the SOA4All EU Project, March 2009.

[KVPM11] Jacek Kopecky, Tomas Vitvar, Carlos Pedrinaci, and Maria Maleshkova. RESTful services with lightweight machine-readable descriptions and semantic annotations. In Erik Wilde and Cesare Pautasso, editors, *REST: From Research to Practice*, pages 473–506. Springer, 2011.

[KZZ11] P. Knoth, L. Zilka, and Z. Zdrahal. Using explicit semantic analysis for cross-lingual link discovery. In *Workshop: 5th International Workshop on Cross Lingual Information Access: Computational Linguistics and the Information Need of Multilingual Societies (CLIA) at The 5th International Joint Conference on Natural Language Processing*, 2011.

[LÓ99] M. Mariano Fernández López. Overview of Methodologies for Building Ontologies. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem Solving Methods (KRR5) Stockholm, Sweden, August 2, 1999*, 1999.

[LB07] Antonio Lopes and Luis Miguel Botelho. Executing Semantic Web Services with a Context-Aware Service Execution Agent. In Jingshan Huang, Ryszard Kowalczyk, Zakaria Maamar, David L. Martin, Ingo Muller, Suzette Stoutenburg, and Katia P. Sycara, editors, *SOCASE*, volume 4504 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.

[LC07] Sigma On Kee Lee and Andy Hon Wai Chun. Automatic tag recommendation for the web 2.0 blogosphere using collaborative tagging and hybrid ANN semantic structures. In *ACOS'07: Proceedings of the 6th Conference on WSEAS International Conference on Applied Computer Science*, pages 88–93, 2007.

[LHPD12] Chenghua Lin, Yulan He, Carlos Pedrinaci, and John Domingue. Feature LDA: A supervised topic model for automatic detection of Web API documentations

from the web. In Philippe Cudra-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jerome Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2012.

[LJ10] D. Lambert and Domingue. J. Photorealistic semantic web service groundings: unifying RESTful and XML-RPC groundings using rules, with an application to flickr. In *In the 4th International Web Rule Symposium (RULEML)*, 2010.

[LLZ+07] Y. Li, Y. Liu, L. Zhang, G. Li, B. Xie, and J. Sun. An exploratory study of web services on the internet. In *Proceedings of the 6th International Conference on Semantic Web (ISWC)*, pages 380–387, 2007.

[LMK05] J. Luo, B. Montrose, and M. Kang. An approach for semantic query processing with UDDI. *OTM Workshops*, pages 89–98, 2005.

[LMKD11] C. Li, N. andPedrinaci, M. Maleshkova, J. Kopecky, and J. Domingue. OmniVoke: A framework for automating the invocation of Web APIs. In *Proceedings of 5th IEEE International Conference on Semantic Computing*, 2011.

[LMR07] Jonathan Douglas Lathem, John Miller, and Lakshmish Ramaswamy. SA-REST: Bring the power of semantics to REST-based web services. Master's thesis, University of Georgia, August 2007.

[LPK+11] N. Li, C. Pedrinaci, J. Kopecky, M. Maleshkova, D. Liu, and J. Domingue. Towards automated invocation of Web APIs, 2011.

[LRD09] P. Leitner, F. Rosenberg, and S. Dustdar. Daios: Efficient dynamic web service invocation. In *Presented at IEEE Internet Computing*, 2009.

[LRPF04] R. Lara, D. Roman, A. Polleres, and D. Fensel. A conceptual comparison of WSMO and OWL-S. In *Proceedings of the European Conference on Web Services*, 2004.

[MA10] et al M. Atwood. OAuth core 1.0 specification. http://oauth.net/core/1.0/, last viewed June 2010.

[Mar04] D. Martin. OWL-S: Semantic Markup for Web Services. W3C member submission, 22 November 2004. http://www.w3.org/Submission/OWL-S/, 2004.

[MBM+07] D. L. Martin, M. H. Burstein, D. V. McDermott, S. A. McIlraith, et al. Bringing Semantics to Web Services with OWL-S. *WWW Journal of Universal Computer Science*, pages 243–277, 2007.

[MGPD09] M. Maleshkova, L. Gridinoc, C. Pedrinaci, and J Domingue. Supporting the semi-automatic acquisition of semantic RESTful service descriptions. In *Poster session of the European Semantic Web Conference, ESWC*, 2009.

[MKP09] M. Maleshkova, J. Kopecky, and C. Pedrinaci. Adapting SAWSDL for semantic annotations of RESTful services. In *Beyond SAWSDL at OnTheMove Federated Conferences and Workshops*, 2009.

[MPD09a] M. Maleshkova, C. Pedrinaci, and J. Domingue. Semantically annotating RESTful services with SWEET. In *Demo session at 8th International Semantic Web Conference ISWC*, 2009.

[MPD09b] M. Maleshkova, C. Pedrinaci, and J. Domingue. Supporting the creation of semantic RESTful service descriptions. In *Service Matchmaking and Resource Retrieval in the Semantic Web (SMR2) at 8th International Semantic Web Conference*, 2009.

[MPD10a] M. Maleshkova, C. Pedrinaci, and J. Domingue. Investigating Web APIs on the World Wide Web. *European Conference on Web Services (ECOWS)*, 2010.

[MPD+10b] M. Maleshkova, C. Pedrinaci, J. Domingue, G. Alvaro, and I. Martinez. Using semantics for automating the authentication of Web APIs. *International Semantic Web Conference (ISWC)*, 2010. Shanghai, China.

[MPL+11] Maria Maleshkova, Carlos Pedrinaci, Ning Li, Jacek Kopecky, and John Domingue. Lightweight semantics for automating the invocation of Web APIs. In Kwei-Jay Lin, Christian Huemer, M. Brian Blake, and Boualem Benatallah, editors, *SOCA*, pages 1–4. IEEE, 2011.

[MSZ01] S. McIlraith, T. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, 16(2):46–53, 2001.

[MVD12] Jose Antonio Mateo, Valentin Valero, and Gregorio Diaz. BPEL-RF: A formal framework for BPEL orchestrations integrating distributed resources. *CoRR*, abs/1203.1760, 2012.

[MZKP11] Maria Maleshkova, Lukas Zilka, Petr Knoth, and Carlos Pedrinaci. Cross-lingual Web API classification and annotation. In Elena Montiel-Ponsoda, John McCrae, Paul Buitelaar, and Philipp Cimiano, editors, *MSW*, volume 775 of *CEUR Workshop Proceedings*, pages 1–12. CEUR-WS.org, 2011.

[NFM00] N.F. Noy, R.W. Fergerson, and M.A. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility. *Lecture Notes in Computer Science*, 1937:69–82, 2000.

[NHOH04] Shinichi Nagano, Tetsuo Hasegawa, Akihiko Ohsuga, and Shinichi Honiden. Dynamic invocation model of web services using subsumption relations. In *ICWS*, pages 150–. IEEE Computer Society, 2004.

[NKMHB06] N. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker. Web services security: SOAP message security 1.1. *WS-Security 2004*, 2006.

[NPDZ08] B. Norton, C. Pedrinaci, J. Domingue, and M. Zaremba. Semantic execution environments for semantics-enabled SOA. *IT - Methods and Applications of Informatics and Information Technology. Special Issue in Service-Oriented Architectures*, pages 118–121, 2008.

[NV04] R. Navigli and P. Velardi. Learning domain ontologies from document warehouses and dedicated web sites. *Computational Linguistics*, 30(2):151–179, 2004.

[OHE03] Phillipa Oaks, Arthur H. M. Ter Hofstede, and David Edmond. Capabilities: describing what services can do. In *Proceedings of the 1st International Conference on Service Oriented Computing, 15–18*, 2003.

[O'R09] Tim O'Reilly. *What is Web 2.0*. O'Reilly Media, 2009.

[OTSV04] Nicole Oldham, Christopher Thomas, Amit Sheth, and Kunal Verma. METEOR-S Web Service Annotation Framework with Machine Learning Classification. In *Proc. of the 1 st Int. Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, 2004.

[PASS03] Massimo Paolucci, Anupriya Ankolekar, Naveen Srinivasan, and Katia P. Sycara. The DAML-S virtual machine. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003.

[Pau09] C. Pautasso. RESTful web service composition with BPEL for REST. *Data and Knowledge Engineering journal*, pages 68:851–866, 2009.

[PB07] Michael J. Pazzani and Daniel Billsus. Content-based recommendation systems. In *The Adaptive Web: Methods and Strategies of Web Personalization. Volume 4321 of Lecture Notes in Computer Science*, pages 325–341. Springer-Verlag, 2007.

[PD10] C. Pedrinaci and J. Domingue. Toward the next wave of services: Linked services for the web of data. *Journal of Universal Computer Science*, pages 1694–1719, 2010.

[PDS10] C. Pedrinaci, J. Domingue, and A. Sheth. Semantic Web Services. In John Domingue and James A. Hendler, editors, *Handbook of Semantic Web Technologies*, volume 2. Springer, 2010.

[PKPS02] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *First Int. Semantic Web Conf.*, 2002.

[PLM+10] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecky, and J. Domingue. iServe: a linked services publishing platform. *Workshop: Ontology Repositories and Editors for the Semantic Web at 7th Extended Semantic Web Conference*, 2010.

[PMZP12] C. Pedrinaci, M. Maleshkova, M. Zaremba, and M. Panahiazar. Handbook of service description: USDL and its methods, 2012.

[POSV04] Abhijit A. Patil, Swapna A. Oundhakar, Amit P. Sheth, and Kunal Verma. METEOR-S web service annotation framework. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 553–562, New York, NY, USA, 2004. ACM Press.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[PSSN03] M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura. Towards a semantic choreography of web services: From WSDL to DAML-S. In *First International Conference on Web Services (ICWS'03)*, June 2003.

[PTDL08] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(2), 223-255, 2008.

[PZ09] Jinie Pak and Lina Zhou. A framework for ontology evaluation. In Raj Sharman, H. Raghav Rao, and T. S. Raghu, editors, *WEB*, volume 52 of *Lecture Notes in Business Information Processing*, pages 10–18. Springer, 2009.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful web services vs. "big" web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[Ren10] Bruno Renie. Annotation recommendation for the semantic description of RESTful web services. Master's thesis, Ecole Nationale Superieure des Mines, Saint Etienne and Knowledge Media Institute, The Open University, Milton Keynes, 2010.

[Riv10] R. Rivest. The MD5 message-digest algorithm. http://tools.ietf.org/html/rfc1321, last viewed June 2010.

[RKD+09] Florian Rosenberg, Rania Khalaf, Matthew J. Duftler, Francisco Curbera, and Paula Austel. End-to-end security for enterprise mashups. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *ICSOC/*, volume 5900 of *Lecture Notes in Computer Science*, pages 389–403, 2009.

[RKL+05] D. Roman, U. Keller, H. Lausen, J. Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, pages 1(1): 77 – 106, 2005.

[RNBY99] Berthier Ribeiro-Neto and Ricardo Baeza-Yates. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[Rod08] Alex Rodriguez. RESTful Web services: The basics. *IBM developerWorks*, November 2008.

[RR07] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly, May 2007.

[RS04] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In Jorge Cardoso and Amit P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2004.

[RSK12] Dominik Renzel, Patrick Schlebusch, and Ralf Klamma. Today's top RESTful services and why they are not RESTful. In Xiaoyang Sean Wang, Isabel F. Cruz, Alex Delis, and Guangyan Huang, editors, *WISE*, volume 7651 of *Lecture Notes in Computer Science*, pages 354–367. Springer, 2012.

[S+97] David J. Schultz et al. IEEE standard for developing software life cycle processes. IEEE std, IEEE, New York, USA, 1997.

[SAS02] York Sure, J"urgen Angele, and Steffen Staab. OntoEdit: Guiding ontology development by methodology and inferencing. In *CoopIS/DOA/ODBASE*, pages 1205–1222, 2002.

[SBC10] Victor Saquicela, Luis Manuel Vilches Bláquez, and Oscar Corcho. Semantic annotation of RESTful services using external resources. In Florian Daniel and Federico Michele Facca, editors, *ICWE Workshops*, volume 6385 of *Lecture Notes in Computer Science*, pages 266–276. Springer, 2010.

[SBC11] Victor Saquicela, Luis Manuel Vilches Blázquez, and Oscar Corcho. Lightweight semantic annotation of geospatial RESTful services. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia,

Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *ESWC (2)*, volume 6644 of *Lecture Notes in Computer Science*, pages 330–344. Springer, 2011.

[SBC12] Victor Saquicela, Luis Manuel Vilches Blázquez, and Oscar Corcho. Adding semantic annotations into (geospatial) RESTful services. In *Int. J. Semantic Web Inf. Syst.*, volume 8, pages 51–71, 2012.

[SC08] P. Sorg and P. Cimiano. Cross-lingual information retrieval with explicit semantic analysis. In *In Working Notes for the CLEF Workshop*, 2008.

[SET09] Toby Segaran, Colin Evans, and Jamie Taylor. *Programming the Semantic Web*. O'Reilly, Beijing, 2009.

[SGL07] A. P. Sheth, K. Gomadam, and J. Lathem. SA-REST: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6):91-94, 2007.

[SH07] Sanjay C. Sood and Kristian J. Hammond. TagAssist: Automatic Tag Suggestion for Blog Posts. In *In International Conference on Weblogs and Social*, 2007.

[SH10] S. Speiser and A. Harth. Taking the lids off data silos. In *In I-SEMANTICS*, 2010.

[SHBL06] N. Shadbolt, W. Hall, and T. Berners-Lee. The Semantic Web Revisited. *IEEE Intelligent Systems*, 2006.

[She03] Amit Sheth. Semantic Web Process Lifecycle: Role of Semantics in Annotation, Discovery, Composition and Orchestration. Invited Talk at WWW 2003 Workshop on E-Services and the Semantic Web, May 2003.

[SHJJ09] H. Story, B. Harbulot, I. Jacobi, and M. Jones. FOAF+SSL: RESTful authentication for the Social Web. In *In SPOT2009 European Semantic Web Conference*, 2009.

[SIS+11] Manu Sporny, Toby Inkster, Henry Story, Bruno Harbulot, and Reto Bachmann-Gmür. Web Identification and Discovery, February 2011.

[SMP10] Marta Sabou, Maria Maleshkova, and Jeff Pan. Semantically enabling web service repositories, 2010.

[SMSV05] Kaarthik Sivashanmugam, John A. Miller, Amit P. Sheth, and Kunal Verma. Framework for semantic web process composition, 2005.

[SP04] Evren Sirin and Bijan Parsia. The OWL-S Java API. In *Proceedings of the Third International Semantic Web Conference. 2004. References*, 2004.

[SSS06] York Sure, Steffen Staab, and Rudi Studer. Ontology engineering methodologies. *In Semantic Web Technologies: Trends and Research in Ontology-based Systems*, 2006.

[Suz03] Satoshi Suzuki. Probabilistic word vector and similarity based on dictionaries. In Alexander F. Gelbukh, editor, *CICLing*, volume 2588 of *Lecture Notes in Computer Science*, pages 562–572. Springer, 2003.

[SVSM03] Kaarthik Sivashanmugam, Kunal Verma, Amit Sheth, and John Miller. Adding semantics to web services standards. In *Proceedings of the 2003 International Conference on Web Services (ICWS'03)*, pages 395–401, Las Vegas, NV, June 2003.

[SWGS05] M. Sabou, C. Wroe, C. Goble, and H. Stuckenschmidt. Learning domain ontologies for semantic web service descriptions. *Journal of Web Semantics*, 3(4), 2005.

[TDO07] Giovanni Tummarello, Renaud Delbru, and Eyal Oren. Sindice.com: Weaving the open linked data. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2007.

[TH01] Loren Terveen and Will Hill. Beyond recommender systems: Helping people help each other. In *HCI in the New Millennium*, pages 487–509. Addison-Wesley, 2001.

[The12a] The Apache Software Foundation. Apache Axis 2. http://ws.apache.org/axis2/, April 2012.

[The12b] The Apache Software Foundation. Apache CXF: An open source service framework. http://incubator.apache.org/cxf/, 2012.

[TK01] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.

[TKSA12] Mohsen Taheriyan, Craig A. Knoblock, Pedro A. Szekely, and José Luis Ambite. Rapidly integrating services into the linked data cloud. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérome Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *International Semantic Web Conference (1)*, volume 7649 of *Lecture Notes in Computer Science*, pages 559–574. Springer, 2012.

[VKVF08] T. Vitvar, J. Kopecky, J. Viskova, and D. Fensel. WSMOLite annotations for web services. *5th European Semantic Web Conference, ESWC 2008*, In The Semantic Web: Research and Applications:Springer, 2008.

[Vra10] Denny Vrandečić. *Ontology Evaluation*. PhD thesis, Karlsruhe Institute of Technology, 2010.

[VSD+12] Ruben Verborgh, Thomas Steiner, Davy Van Deursen, Sam Coppens, Joaquim Gabarró Vallés, and Rik Van de Walle. Functional descriptions as the bridge between hypermedia APIs and the Semantic Web. In Rosa Alarcón, Cesare Pautasso, and Erik Wilde, editors, *WS-REST*, pages 33–40. ACM, 2012.

[W3C01] W3C WS Description Working Group. Web service description language (WSDL) Version 1.1, W3C Note. http://www.w3.org/TR/wsdl, March 2001.

[W3C07a] W3C WS Description Working Group. Web service description language (WSDL) Version 2.0, W3C proposed recommendation. http://www.w3.org/TR/wsdl20-primer, May 2007.

[W3C07b] W3C WS XML Protocol Working Group. Simple Object Access Protocol (SOAP) Version 1.2. http://www.w3.org/TR/soap/, April 2007.

[WL02] M. D. Wilkinson and M. Links. BioMOBY: An open source biological web services proposal. *Briefings in Bioinformatics*, 3:331–341, 2002.

[Wor07] B. Worthen. Mashups sew data together: Mashup tools can cut costs, time for linking information sources. *The Wallstreet Journal*, 31 July, 2007.

[ZN08] Amal Zouaq and Roger Nkambou. Building domain ontologies from text for educational purposes. *IEEE Transactions on Learning Technologies*, 1:49–62, 2008.